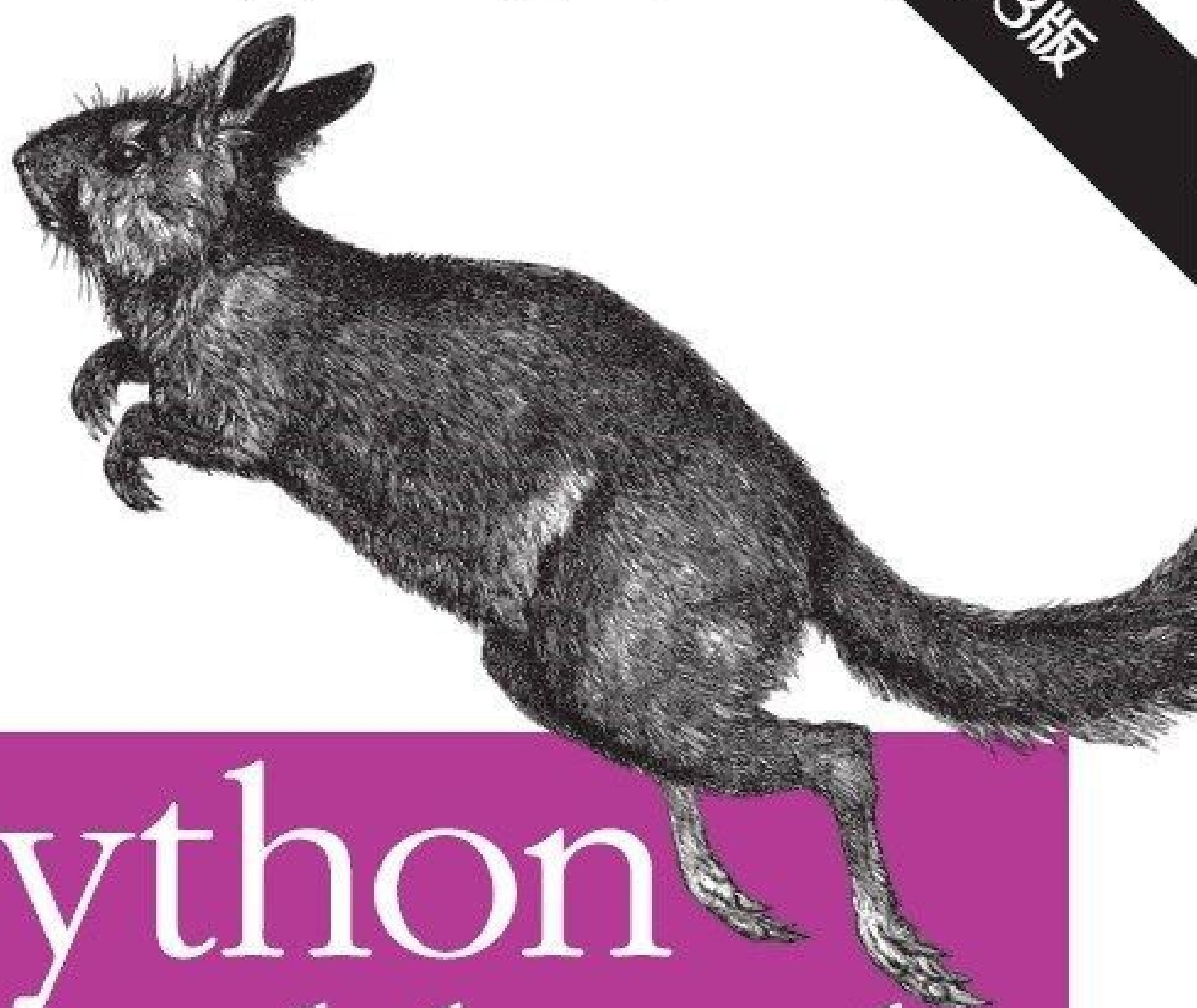
Python Cookbook

Recipes for Mastering Python 3

第3版



Python Cookbook™

中文版

[美] David Beazley & Brian K. Jones 著

陈舸 译

O'REILLY®

人民邮电出版社

POSTS & TELECOM PRESS

目　录

版权信息

书名：Python Cookbook（第3版）中文版

ISBN：978-7-115-37959-7

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

- 著　　[美] David Beazley Brian K.Jones

　译　　陈　舸

责任编辑　傅道坤

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

Copyright © 2013 by O’Reilly Media.Inc.

Simplified Chinese Edition, jointly published by O’Reilly Media, Inc. and Posts & Telecom Press, 2015. Authorized translation of the English edition, 2011 O’Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体字版由O’Reilly Media, Inc.授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

内容提要

本书介绍了Python应用在各个领域中的一些使用技巧和方法，其主题涵盖了数据结构和算法，字符串和文本，数字、日期和时间，迭代器和生成器，文件和I/O，数据编码与处理，函数，类与对象，元编程，模块和包，网络和Web编程，并发，实用脚本和系统管理，测试、调试以及异常，C语言扩展等。

本书覆盖了Python应用中的很多常见问题，并提出了通用的解决方案。书中包含了大量实用的编程技巧和示例代码，并在Python 3.3环境下进行了测试，可以很方便地应用到实际项目中去。此外，本书还详细讲解了解决方案是如何工作的，以及为什么能够工作。

本书非常适合具有一定编程基础的Python程序员阅读参考。

O’Reilly Media, Inc. 介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978 年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势—通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

关于作者

David Beazley 是一位居住在芝加哥的独立软件开发者以及图书作者。他主要的工作在于编程工具，提供定制化的软件开发服务，以及为软件开发者、科学家和工程师教授编程实践课程。他最为人熟知的工作在于Python编程语言，他已为此创建了好几个开源的软件包（例如Swig和PLY），并且是备受赞誉的图书Python Essential Reference的作者。他也对C、C++以及汇编语言下的系统编程有着丰富的经验。

Brain K. Jones 是普林斯顿大学计算机系的一位系统管理员。

前言

自2008年以来，我们已经目睹了整个Python世界正缓慢向着Python 3进化的事实。众所周知，完全接纳Python 3要花很长的时间。事实上，就在写作本书时（2013年），大多数Python程序员仍然坚持在生产环境中使用Python 2。关于Python 3不能向后兼容的事实也已经做了许多努力来补救。的确，向后兼容性对于任何已经存在的代码库来说是个问题。但是，如果你着眼于未来，你会发现Python 3带来的好处绝非那么简单。

正因为Python 3是着眼于未来的，本书在之前的版本上做了很大程度的修改。首先也是最重要的一点，这是一本积极拥抱Python 3的书。所有的章节都采用Python 3.3来编写并进行了验证，没有考虑老的Python版本或者“老式”的实现方式。事实上，许多章节都只适用于Python 3.3甚至更高的版本。这么做可能会有风险，但是最终的目的是要编写一本Python 3的秘籍，尽可能基于最先进的工具和惯用法。我们希望本书可以指导人们用Python 3编写新的代码，或者帮助开发人员将已有的代码升级到Python 3。

无需赘言，以这种风格来编写本书给编辑工作

带来了一定的挑战。只要在网络上搜索一下Python秘籍，立刻就能在ActiveState的Python版块或者Stack Overflow这样的站点上找到数以千计的使用心得和秘籍。但是，大部分这类资源已经沉浸在历史和过去中了。由于这些心得和秘籍几乎完全是针对Python 2所写的，其中常常包含有各种针对Python不同版本（例如2.3版对比2.4版）之间差异的变通方法和技巧。此外，这些网上资源常常使用过时的技术，而这些技术现在成了Python 3.3的内建功能。想寻找专门针对Python 3的资源会比较困难。

本书并非搜寻特定于Python 3方面的秘籍将其汇集而成，本书的主题都是在创作中由现有的代码和技术而产生出的灵感。我们将这些思想作为跳板，尽可能采用最现代化的Python编程技术来写作，因此本书的内容完全是原创性的。对于任何希望以现代化的风格来编写代码的人，本书都可以作为参考手册。

在选择应该包含哪些章节时，我们有一个共识。那就是根本不可能编写一本涵盖了每种Python用途的书。因此，我们在主题上优先考虑Python语言核心方面的内容，以及能够广泛适用于各种应用领域的常见任务。此外，有许多秘籍是用来说明在Python 3中新增的功能，这对许多人来说比较陌

生，甚至对于那些使用老版Python经验丰富的程序员也是如此。我们也会优先选择普遍适用的编程技术（即，编程模式）作为主题，而不会选择那些试图解决一个非常具体的实际问题但适用范围太窄的内容。尽管在部分章节中也提到了特定的第三方软件包，但本书绝大多数章节都只关注语言核心和标准库。

本书适合谁

本书的目标读者是希望加深对Python语言的理解以及学习现代化编程惯用法的有经验的程序员。本书许多内容把重点放在库、框架和应用中使用的高级技术上。本书假设读者已经有了理解本书主题的必要背景知识（例如对计算机科学的一般性知识、数据结构、复杂度计算、系统编程、并发、C语言编程等）。此外，本书中提到的秘籍往往只是一个框架，意在提供必要的信息让读者可以起步，但是需要读者自己做更多的研究来填补其中的细节。因此，我们假设读者知道如何使用搜索引擎以及优秀的Python在线文档。

有一些更加高级的章节将作为读者耐心阅读的奖励。这些章节对于理解Python底层的工作原理提供了深刻的见解。你将学到新的技巧和技术，可以将这些知识运用到自己的代码中去。

本书不适合谁

这不是一本用来给初学者首次学习Python编程而使用的书。事实上，本书已经假设读者通过Python教程或者入门书籍了解了基本知识。本书同样不能用来作为快速参考手册（即，快速查询特定模块中的某个函数）。相反，本书的目标是把重点放在特定的编程主题上，展示可能的解决方案并以此作为跳板引导读者学习更加高级的内容。这些内容你可能会在网上或者参考书中遇到过。

本书中的约定



提示

这个图标用来强调一个提示、建议或一般说明。



警告

这个图标用来说明一个警告或注意事项。

在线代码示例

本书中几乎所有的代码示例都可以

在http://github.com/dabeaz/python-cookbook 上找到。作者欢迎读者针对代码示例提供bug修正、改进以及评论。

使用代码示例

本书的目的是为了帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们打个招呼。例如，你可以在无需获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发O’Reilly图书中的代码光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的属性信息是最好不过。一个属性信息通常包括书名、作者、出版社和ISBN。例如：Python Cookbook, 3rd edtion, by David Beazley and Brain K.Jones(O’Reilly)。Copyright 2013 David Beazley and Brain Jones, 978-1-449-34037-7。

在使用书中的代码时，如果不确定是否属于正

常使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O’Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。你可以通过链接 http://oreil.ly/python_cookbook_3e 来访问页面。

关于本书的技术性问题或建议，请发邮件到：

bookquestions@oreilly.com

欢迎登录我们的网站（http://www.oreilly.com），查看更多我们的书籍、课程、会议和最新动态等信息。

Facebook: http://facebook.com/oreilly

Twitter: http://twitter.com/oreillymedia

YouTube: http://www.youtube.com/oreillymedia

致谢

我们要感谢本书的技术校审人员，他们是Jake Vanderplas、Robert Kern以及Andrea Crotti。感谢他们非常有用的评价，也要感谢整个Python社区的支持和鼓励。我们也要感谢本书第2版的编辑Alex Martelli、Anna Ravenscroft以及David Ascher。尽管本书的第3版是新创作的，但之前的版本为本书提供了挑选主题以及所感兴趣的秘籍的初始框架。最后也是最重要的是，我们要感谢本书早期版本的读者，感谢你们为本书的改进做出的评价和建议。

David Beazley的致谢

写一本书绝非易事。因此，我要感谢我的妻子Paula以及我的两个儿子，感谢你们的耐心以及支持。本书中的许多素材都来自于我过去6年里所教的与Python相关的训练课程。因此，我要感谢所有参加了我的课程的学生，正是你们最终促成了本书的问世。我也要感谢Ned Batchelder、Travis Oliphant、Peter Wang、Brain Van de Ven、Hugo Shi、Raymond Hettinger、Michael Foord以及Daniel Klein，感谢他们飞到世界各地去教学，而让我可以留在芝加哥的家中完成本书的写作。感谢来自O’Reilly的Meghan Blanchette以及Rachel Roumeliotis，你们见证了本书的创作过程，当然也经历了那些无法预料到的延期。最后也是最重要的是，我要感谢Python社区不间断的支持，以及容忍我那不着调的胡思乱想。

David M.Beazley

http://www.dabeaz.com

https://twitter.com/dabeaz

Brain Jones的致谢

我要感谢我的合著者David Beazley以及O’Reilly的Meghan Blanchette和Rachel Roumeliotis，感谢你们和我一起完成了本书的创作。我也要感谢我的妻子Natasha，感谢你在我写作本书时给予的耐心和鼓励，也要谢谢你对于我所有追求的支持。我尤其要感谢Python社区。虽然我已经在多个开源项目和编程语言中有所贡献，但与Python社区长久以来所做的如此令人欣慰和富有意义的工作相比，我做的算不上什么。

Brain K.Jones

http://www.protocolostomy.com

https://twitter.com/bkjones

第1章　数据结构和算法

Python内置了许多非常有用的数据结构，比如列表（list）、集合（set）以及字典（dictionary）。就绝大部分情况而言，我们可以直接使用这些数据结构。但是，通常我们还需要考虑比如搜索、排序、排列以及筛选等这一类常见的问题。因此，本章的目的就是来讨论常见的数据结构和同数据有关的算法。此外，在collections模块中也包含了针对各种数据结构的解决方案。

1.1　将序列分解为单独的变量

1.1.1　问题

我们有一个包含 N 个元素的元组或序列，现在想将它分解为N个单独的变量。

1.1.2　解决方案

任何序列（或可迭代的对象）都可以通过一个简单的赋值操作来分解为单独的变量。唯一的要求是变量的总数和结构要与序列相吻合。例如：

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
```

```
2012
>>> mon
12
>>> day
21
>>>
```

如果元素的数量不匹配，将得到一个错误提示。例如：

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

1.1.3 讨论

实际上不仅仅只是元组或列表，只要对象恰好是可迭代的，那么就可以执行分解操作。这包括字符串、文件、迭代器以及生成器。比如：

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

当做分解操作时，有时候可能想丢弃某些特定的值。Python并没有提供特殊的语法来实现这一点，但是通常可以选一个用不到的变量名，以此来作为要丢弃的值的名称。例如：

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

但是请确保选择的变量名没有在其他地方用到过。

1.2　从任意长度的可迭代对象中分解元素

1.2.1　问题

需要从某个可迭代对象中分解出N个元素，但是这个可迭代对象的长度可能超过N，这会导致出现“分解的值过多（too many values to unpack）”的异常。

1.2.2　解决方案

Python的“*表达式”可以用来解决这个问题。例如，假设开设了一门课程，并决定在期末的作业成绩中去掉第一个和最后一个，只对中间剩下的成绩做平均分统计。如果只有4个成绩，也许可以简单地将4个都分解出来，但是如果有24个呢？*表达式使这一切都变得简单：

```
def

drop_first_last(grades):
    first, *middle, last = grades
    return
```

```
avg(middle)
```

另一个用例是假设有一些用户记录，记录由姓名和电子邮件地址组成，后面跟着任意数量的电话号码。则可以像这样分解记录：

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-5
>>> name, email, *phone_numbers = user_record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

不管需要分解出多少个电话号码（甚至没有电话号码），变量phone_numbers都一直是列表，而这是毫无意义的。如此一来，对于任何用到了变量phone_numbers的代码都不必对它可能不是一个列表的情况负责，或者额外做任何形式的类型检查。

由*修饰的变量也可以位于列表的第一个位置。例如，比方说用一系列的值来代表公司过去8个季度的销售额。如果想对最近一个季度的销售额同前7个季度的平均值做比较，可以这么做：

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return


avg_comparison(trailing_avg, current_qtr)
```

从Python解释器的角度来看，这个操作是这样的：

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

1.2.3 讨论

对于分解未知或任意长度的可迭代对象，这种扩展的分解操作可谓是量身定做的工具。通常，这类可迭代对象中会有一些已知的组件或模式（例如，元素1之后的所有内容都是电话号码），利用*表达式分解可迭代对象使得开发者能够轻松利用这些模式，而不必在可迭代对象中做复杂花哨的操作才能得到相关的元素。

*式的语法在迭代一个变长的元组序列时尤其

有用。例如，假设有一个带标记的元组序列：

```
records = [
     ('foo', 1, 2),
      ('bar', 'hello'),
      ('foo', 3, 4),
]

def


do_foo(x, y):
    print


('foo', x, y)

def


do_bar(s):
    print


('bar', s)

for


tag, *args in


records:
    if


tag == 'foo':
        do_foo(*args)
elif


tag == 'bar':
        do_bar(*args)
```

当和某些特定的字符串处理操作相结合，比如做拆分（splitting）操作时，这种*式的语法所支持的分解操作也非常有用。例如：

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

有时候可能想分解出某些值然后丢弃它们。在分解的时候，不能只是指定一个单独的*，但是可以使用几个常用来表示待丢弃值的变量名，比如_或者ign（ignored）。例如：

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

*分解操作和各种函数式语言中的列表处理功能有着一定的相似性。例如，如果有一个列表，可以像下面这样轻松将其分解为头部和尾部：

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

在编写执行这类拆分功能的函数时，人们可以假设这是为了实现某种精巧的递归算法。例如：

```
>>> def


sum(items):
...


head, *tail = items
...      return


head + sum(tail) if


tail else


head
...
```

```
>>> sum(items)
36
>>>
```

但是请注意，递归真的不算是Python的强项，这是因为其内在的递归限制所致。因此，最后一个例子在实践中没太大的意义，只不过是一点学术上的好奇罢了。

1.3　保存最后N个元素

1.3.1　问题

我们希望在迭代或是其他形式的处理过程中对最后几项记录做一个有限的历史记录统计。

1.3.2　解决方案

保存有限的历史记录可算是collections.deque的完美应用场景了。例如，下面的代码对一系列文本行做简单的文本匹配操作，当发现有匹配时就输出当前的匹配行以及最后检查过的N行文本。

```
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file



if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
```

```
        print(line, end='')
        print('-'*20)
```

1.3.3 讨论

如同上面的代码片段中所做的一样，当编写搜索某项记录的代码时，通常会用到含有yield关键字的生成器函数。这将处理搜索过程的代码和使用搜索结果的代码成功解耦开来。如果对生成器还不熟悉，请参见4.3节。

deque(maxlen=N)创建了一个固定长度的队列。当有新记录加入而队列已满时会自动移除最老的那条记录。例如：

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

尽管可以在列表上手动完成这样的操作（append、del），但队列这种解决方案要优雅得多，运行速度也快得多。

更普遍的是，当需要一个简单的队列结构时，deque可祝你一臂之力。如果不指定队列的大小，也就得到了一个无界限的队列，可以在两端执行添加和弹出操作，例如：

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

从队列两端添加或弹出元素的复杂度都是O(1)。这和列表不同，当从列表的头部插入或移除元素时，列表的复杂度为O(N)。

1.4　找到最大或最小的N个元素

1.4.1　问题

我们想在某个集合中找出最大或最小的N个元素。

1.4.2　解决方案

heapq模块中有两个函数——nlargest()和nsmallest()——它们正是我们所需要的。例如：

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]



print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

这两个函数都可以接受一个参数key，从而允许它们工作在更加复杂的数据结构之上。例如：

```
portfolio = [
   {'name': 'IBM', 'shares': 100, 'price': 91.1},
   {'name': 'AAPL', 'shares': 50, 'price': 543.22},
   {'name': 'FB', 'shares': 200, 'price': 21.09},
   {'name': 'HPQ', 'shares': 35, 'price': 31.75},
   {'name': 'YHOO', 'shares': 45, 'price': 16.35},
   {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda


s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda


s: s['price'])
```

1.4.3 讨论

如果正在寻找最大或最小的N个元素，且同集合中元素的总数目相比，N很小，那么下面这些函数可以提供更好的性能。这些函数首先会在底层将数据转化成列表，且元素会以堆的顺序排列。例如：

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq



>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
```

```
>>>
```

堆最重要的特性就是heap[0]总是最小那个的元素。此外，接下来的元素可依次通过heapq.heappop()方法轻松找到。该方法会将第一个元素（最小的）弹出，然后以第二小的元素取而代之（这个操作的复杂度是O(logN)，N代表堆的大小）。例如，要找到第3小的元素，可以这样做：

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

当所要找的元素数量相对较小时，函数nlargest()和nsmallest()才是最适用的。如果只是简单地想找到最小或最大的元素（N=1时），那么用min()和max()会更加快。同样，如果N和集合本身的大小差不多大，通常更快的方法是先对集合排序，然后做切片操作（例如，使用sorted(items)[:N]或者sorted(items)[-N:]）。应该要注意的是，nlargest()和nsmallest()的实际实现会根据使用它们的方式而有所不同，可能会相应作出一些优化措施（比如，当N的大小同输入大小很接近时，就会采

用排序的方法）。

使用本节的代码片段并不需要知道如何实现堆数据结构，但这仍然是一个有趣也是值得去学习的主题。通常在优秀的算法和数据结构相关的书籍里都能找到堆数据结构的实现方法。在heapq模块的文档中也讨论了底层实现的细节。

1.5 实现优先级队列

1.5.1 问题

我们想要实现一个队列，它能够以给定的优先级来对元素排序，且每次pop操作时都会返回优先级最高的那个元素。

1.5.2 解决方案

下面的类利用heapq模块实现了一个简单的优先级队列：

```
import heapq
class PriorityQueue:

    def __init__(self):
        self._queue = []
        self._index = 0

def push(self, item, priority):
    heapq.heappush(self._queue, (-priority, self._index, item))
    self._index += 1

def pop(self):
    return heapq.heappop(self._queue)[-1]
```

下面是如何使用这个类的例子：

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

请注意观察，第一次执行pop()操作时返回的元素具有最高的优先级。我们也观察到拥有相同优先级的两个元素（foo和grok）返回的顺序同它们插入到队列时的顺序相同。

1.5.3　讨论

上面的代码片段的核心在于heapq模块的使用。函数heapq.heappush()以及heapq.heappop()分别实现将元素从列表_queue中插入和移除，且保证列表中第一个元素的优先级最低（如1.4节所述）。heappop()方法总是返回“最小”的元素，因此这就是

让队列能弹出正确元素的关键。此外，由于push和pop操作的复杂度都是O(logN)，其中N代表堆中元素的数量，因此就算N的值很大，这些操作的效率也非常高。

在这段代码中，队列以元组(-priority, index, item)的形式组成。把priority取负值是为了让队列能够按元素的优先级从高到低的顺序排列。这和正常的堆排列顺序相反，一般情况下堆是按从小到大的顺序排序的。

变量index的作用是为了将具有相同优先级的元素以适当的顺序排列。通过维护一个不断递增的索引，元素将以它们入队列时的顺序来排列。但是，index在对具有相同优先级的元素间做比较操作时同样扮演了重要的角色。

为了说明Item实例是没法进行次序比较的，我们来看下面这个例子：

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

如果以元组(priority, item)的形式来表示元素，那么只要优先级不同，它们就可以进行比较。但是，如果两个元组的优先级值相同，做比较操作时还是会像之前那样失败。例如：

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

通过引入额外的索引值，以(prioroty, index, item)的方式建立元组，就可以完全避免这个问题。因为没有哪两个元组会有相同的index值（一旦比较操作的结果可以确定，Python就不会再去比较剩下的元组元素了）：

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

如果想将这个队列用于线程间通信，还需要增加适当的锁和信号机制。请参见12.3节的示例学习如何去做。

关于堆的理论和实现在heapq模块的文档中有着详细的示例和相关讨论。

1.6　在字典中将键映射到多个值上

1.6.1　问题

我们想要一个能将键（key）映射到多个值的字典（即所谓的一键多值字典[multidict]）。

1.6.2　解决方案

字典是一种关联容器，每个键都映射到一个单独的值上。如果想让键映射到多个值，需要将这多个值保存到另一个容器如列表或集合中。例如，可能会像这样创建字典：

```
d = {
   'a' : [1, 2, 3],
   'b' : [4, 5]
}

e = {
   'a' : {1, 2, 3},
   'b' : {4, 5}
}
```

要使用列表还是集合完全取决于应用的意图。如果希望保留元素插入的顺序，就用列表。如果希望消除重复元素（且不在意它们的顺序），就用集

合。

为了能方便地创建这样的字典，可以利用collections模块中的defaultdict类。defaultdict的一个特点就是它会自动初始化第一个值，这样只需关注添加元素即可。例如：

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...
```

关于defaultdict，需要注意的一个地方是，它会自动创建字典表项以待稍后的访问（即使这些表项当前在字典中还没有找到）。如果不想要这个功能，可以在普通的字典上调用setdefault()方法来取代。例如：

```
d = {} # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...
```

然而，许多程序员觉得使用setdefault()有点不自然——更别提每次调用它时都会创建一个初始值的新实例了（例子中的空列表[]）。

1.6.3　讨论

原则上，构建一个一键多值字典是很容易的。但是如果试着自己对第一个值做初始化操作，这就会变得很杂乱。例如，可能会写下这样的代码：

```
d = {}
for


key, value in


pairs:
    if


key not in


d:
         d[key] = []
    d[key].append(value)
```

使用defaultdict后代码会清晰得多：

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

这一节的内容同数据处理中的记录归组问题有很强的关联。请参见1.15节的示例。

1.7　让字典保持有序

1.7.1　问题

我们想创建一个字典，同时当对字典做迭代或序列化操作时，也能控制其中元素的顺序。

1.7.2　解决方案

要控制字典中元素的顺序，可以使用collections模块中的OrderedDict类。当对字典做迭代时，它会严格按照元素初始添加的顺序进行。例如：

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"



for key in d:
print(key, d[key])
```

当想构建一个映射结构以便稍后对其做序列化或编码成另一种格式时，OrderedDict就显得特别有用。例如，如果想在进行JSON编码时精确控制各字段的顺序，那么只要首先在OrderedDict中构建数据就可以了。

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

1.7.3　讨论

OrderedDict内部维护了一个双向链表，它会根据元素加入的顺序来排列键的位置。第一个新加入的元素被放置在链表的末尾。接下来对已存在的键做重新赋值不会改变键的顺序。

请注意OrderedDict的大小是普通字典的2倍多，这是由于它额外创建的链表所致。因此，如果打算构建一个涉及大量OrderedDict实例的数据结构（例如从CSV文件中读取100000行内容到OrderedDict列表中），那么需要认真对应用做需求分析，从而判断使用OrderedDict所带来的好处是否能超越因额外的内存开销所带来的缺点。

1.8 与字典有关的计算问题

1.8.1 问题

我们想在字典上对数据执行各式各样的计算（比如求最小值、最大值、排序等）。

1.8.2 解决方案

假设有一个字典在股票名称和对应的价格间做了映射：

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

为了能对字典内容做些有用的计算，通常会利用zip()将字典的键和值反转过来。例如，下面的代码会告诉我们如何找出价格最低和最高的股票。

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')
```

```
max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

同样，要对数据排序只要使用zip()再配合sorted()就可以了，比如：

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),



#                 (45.23, 'ACME'), (205.55, 'IBM'),



#                (612.78, 'AAPL')]
```

当进行这些计算时，请注意zip()创建了一个迭代器，它的内容只能被消费一次。例如下面的代码就是错误的：

```
prices_and_names = zip(prices.values(), prices.keys())
print
```

```
(min(prices_and_names))    # OK

print

(max(prices_and_names))    # ValueError: max() arg is an empty se
```

1.8.3 讨论

如果尝试在字典上执行常见的数据操作，将会发现它们只会处理键，而不是值。例如：

```
min(prices)     # Returns 'AAPL'
max(prices)     # Returns 'IBM'
```

这很可能不是我们所期望的，因为实际上我们是尝试对字典的值做计算。可以利用字典的values()方法来解决这个问题：

```
min(prices.values())    # Returns 10.75

max(prices.values())    # Returns 612.78
```

不幸的是，通常这也不是我们所期望的。比如，我们可能想知道相应的键所关联的信息是什么（例如哪支股票的价格最低？）

如果提供一个key参数传递给min()和max()，就能得到最大值和最小值所对应的键是什么。例如：

```
min(prices, key=lambda

k: prices[k])    # Returns 'FB'


max(prices, key=lambda

k: prices[k])    # Returns 'AAPL'
```

但是，要得到最小值的话，还需要额外执行一次查找。例如：

```
min_value = prices[min(prices, key=lambda
```

```
k: prices[k])]
```

利用了zip()的解决方案是通过将字典的键-值对“反转”为值-键对序列来解决这个问题的。

当在这样的元组上执行比较操作时，值会先进行比较，然后才是键。这完全符合我们的期望，允许我们用一条单独的语句轻松的对字典里的内容做整理和排序。

应该要注意的是，当涉及（value, key）对的比较时，如果碰巧有多个条目拥有相同的value值，那么此时key将用来作为判定结果的依据。例如，在计算min()和max()时，如果碰巧value的值相同，则将返回拥有最小或最大key值的那个条目。示例如下：

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9　在两个字典中寻找相同点

1.9.1　问题

有两个字典，我们想找出它们中间可能相同的地方（相同的键、相同的值等）。

1.9.2　解决方案

考虑如下两个字典：

```
a = {
   'x' : 1,
   'y' : 2,
   'z' : 3
}

b = {
   'w' : 10,
   'x' : 11,
   'y' : 2
}
```

要找出这两个字典中的相同之处，只需通过keys()或者items()方法执行常见的集合操作即可。例如：

```
# Find keys in common
```

```
a.keys() & b.keys() # { 'x', 'y' }

# Find keys in a that are not in b
a.keys() - b.keys() # { 'z' }

# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

这些类型的操作也可用来修改或过滤掉字典中的内容。例如，假设想创建一个新的字典，其中会去掉某些键。下面是使用了字典推导式的代码示例：

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for


key in


a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

1.9.3 讨论

字典就是一系列键和值之间的映射集合。字典的keys()方法会返回keys-view对象，其中暴露了所有的键。关于字典的键有一个很少有人知道的特性，那就是它们也支持常见的集合操作，比如求并

集、交集和差集。因此，如果需要对字典的键做常见的集合操作，那么就能直接使用keys-view对象而不必先将它们转化为集合。

字典的items()方法返回由(key,value)对组成的items-view对象。这个对象支持类似的集合操作，可用来完成找出两个字典间有哪些键值对有相同之处的操作。

尽管类似，但字典的values()方法并不支持集合操作。部分原因是因为在字典中键和值是不同的，从值的角度来看并不能保证所有的值都是唯一的。单这一条原因就使得某些特定的集合操作是有问题的。但是，如果必须执行这样的操作，还是可以先将值转化为集合来实现。

1.10　从序列中移除重复项且保持元素间顺序不变

1.10.1　问题

我们想去除序列中出现的重复元素，但仍然保持剩下的元素顺序不变。

1.10.2　解决方案

如果序列中的值是可哈希（hashable）的，那么这个问题可以通过使用集合和生成器轻松解决。示例如下[1]：

```
def

dedupe(items):
    seen = set()
    for

item in

items:
        if

item not in
```

```
seen:
            yield


item
            seen.add(item)
```

这里是如何使用这个函数的例子：

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

只有当序列中的元素是可哈希的时候才能这么做。如果想在不可哈希的对象（比如列表）序列中去除重复项，需要对上述代码稍作修改：

```
def


dedupe(items, key=None):
    seen = set()
    for


item in


items:
        val = item if
```

```
key is

None else

key(item)
        if

val not in

seen:
            yield

item
            seen.add(val)
```

这里参数key的作用是指定一个函数用来将序列中的元素转换为可哈希的类型，这么做的目的是为了检测重复项。它可以像这样工作：

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2,
>>> list(dedupe(a, key=lambda

d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda

d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

如果希望在一个较复杂的数据结构中，只根据对象的某个字段或属性来去除重复项，那么后一种解决方案同样能完美工作。

1.10.3　讨论

如果想要做的只是去除重复项，那么通常足够简单的办法就是构建一个集合。例如：

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

但是这种方法不能保证元素间的顺序不变[2]，因此得到的结果会被打乱。前面展示的解决方案可避免出现这个问题。

本节中对生成器的使用反映出一个事实，那就是我们可能会希望这个函数尽可能的通用——不必绑定在只能对列表进行处理。比如，如果想读一个文件，去除其中重复的文本行，可以只需这样处理：

```
with

open(somefile,'r') as

f:
    for

line in

dedupe(f):
        ...
```

我们的dedupe()函数也模仿了内置函数sorted()、min()以及max()对key函数的使用方式。例子可参考1.8节和1.13节。

1.11 对切片命名

1.11.1 问题

我们的代码已经变得无法阅读，到处都是硬编码的切片索引，我们想将它们清理干净。

1.11.2 解决方案

假设有一些代码用来从字符串的固定位置中取出具体的数据（比如从一个平面文件或类似的格式）[3]：

```
######    01234567890123456789012345678901234567890123456789012345678901234
record = '....................100 .......513.25 ..........'
cost = int(record[20:32]) * float(record[40:48])
```

与其这样做，为什么不对切片命名呢？

```
SHARES = slice(20,32)
PRICE = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

在后一种版本中，由于避免了使用许多神秘难懂的硬编码索引，我们的代码就变得清晰了许多。

1.11.3 讨论

作为一条基本准则，代码中如果有很多硬编码的索引值，将导致可读性和可维护性都不佳。例如，如果一年以后再回过头来看代码，你会发现自己很想知道当初编写这些代码时自己在想些什么。前面展示的方法可以让我们对代码的功能有着更加清晰的认识。

一般来说，内置的slice()函数会创建一个切片对象，可以用在任何允许进行切片操作的地方。例如：

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del


items[a]
>>> items
[0, 1, 4, 5, 6]
```

如果有一个slice对象的实例s，可以分别通过s.start、s.stop以及s.step属性来得到关于该对象的信息。例如：

```
>>> a = slice(
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

此外，可以通过使用indices(size)方法将切片映射到特定大小的序列上。这会返回一个(start, stop, step)元组，所有的值都已经恰当地限制在边界以内（当做索引操作时可避免出现IndexError异常）。例如：

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for


i in


range(*a.indices(len(s))):
... print
```

```
(s[i])
...


w
r
d
>>>
```

1.12　找出序列中出现次数最多的元素

1.12.1　问题

我们有一个元素序列，想知道在序列中出现次数最多的元素是什么。

1.12.2　解决方案

collections模块中的Counter类正是为此类问题所设计的。它甚至有一个非常方便的most_common()方法可以直接告诉我们答案。

为了说明用法，假设有一个列表，列表中是一系列的单词，我们想找出哪些单词出现的最为频繁。下面是我们的做法：

```
words = [
   'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
   'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around',
   'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'int
   'my', 'eyes', "you're", 'under'
]

from collections import
```

```
Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print


(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

1.12.3　讨论

可以给Counter对象提供任何可哈希的对象序列作为输入。在底层实现中，Counter是一个字典，在元素和它们出现的次数间做了映射。例如：

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

如果想手动增加计数，只需简单地自增即可：

```
>>> morewords = ['why','are','you','not','looking','in','my','eye
>>> for


word in
```

```
morewords:
...


word_counts[word] += 1
...



>>> word_counts['eyes']
9
>>>
```

另一种方式是使用update()方法。

```
>>> word_counts.update(morewords)
>>>
```

关于Counter对象有一个不为人知的特性，那就是它们可以轻松地同各种数学运算操作结合起来使用。例如：

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'ar
         "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'y
         'my': 1, 'why': 1})

>>> # Combine counts
```

```
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'no
         'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
         'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts



>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'ar
         "you're": 1, "don't": 1, 'under': 1})
>>>
```

不用说，当面对任何需要对数据制表或计数的问题时，Counter对象都是你手边的得力工具。比起利用字典自己手写算法，更应该采用这种方式完成任务。

1.13 通过公共键对字典列表排序

1.13.1 问题

我们有一个字典列表，想根据一个或多个字典中的值来对列表排序。

1.13.2 解决方案

利用operator模块中的itemgetter函数对这类结构进行排序是非常简单的。假设通过查询数据库表项获取网站上的成员列表，我们得到了如下的数据结构：

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

根据所有的字典中共有的字段来对这些记录排序是非常简单的，示例如下：

```
from operator import itemgetter
```

```
rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

以上代码的输出为：

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

itemgetter()函数还可以接受多个键。例如下面这段代码：

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print


(rows_by_lfname)
```

这会产生如下的输出：

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
```

```
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

1.13.3 讨论

在这个例子中，rows被传递给内建的sorted()函数，该函数接受一个关键字参数key。这个参数应该代表一个可调用对象（callable），该对象从rows中接受一个单独的元素作为输入并返回一个用来做排序依据的值。itemgetter()函数创建的就是这样一个可调用对象。

函数operator.itemgetter()接受的参数可作为查询的标记，用来从rows的记录中提取出所需要的值。它可以是字典的键名称、用数字表示的列表元素或是任何可以传给对象的__getitem__()方法的值。如果传多个标记给itemgetter()，那么它产生的可调用对象将返回一个包含所有元素在内的元组，然后sorted()将根据对元组的排序结果来排列输出结果。如果想同时针对多个字段做排序（比如例子中的姓和名），那么这是非常有用的。

有时候会用lambda表达式来取代itemgetter()的功能。例如：

```
rows_by_fname = sorted(rows, key=lambda


r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda


r: (r['lname'],r['fname']))
```

这种解决方案通常也能正常工作。但是用itemgetter()通常会运行得更快一些。因此如果需要考虑性能问题的话，应该使用itemgetter()。

最后不要忘了本节中所展示的技术同样适用于min()和max()这样的函数。例如：

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14　对不原生支持比较操作的对象排序

1.14.1　问题

我们想在同一个类的实例之间做排序，但是它们并不原生支持比较操作。

1.14.2　解决方案

内建的sorted()函数可接受一个用来传递可调用对象（callable）的参数key，而该可调用对象会返回待排序对象中的某些值，sorted则利用这些值来比较对象。例如，如果应用中有一系列的User对象实例，而我们想通过user_id属性来对它们排序，则可以提供一个可调用对象将User实例作为输入然后返回user_id。示例如下：

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
```

```
[User(3), User(23), User(99)]
>>>
```

除了可以用lambda表达式外，另一种方式是使用operator.attrgetter()。

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

1.14.3 讨论

要使用lambda表达式还是attrgetter()或许只是一种个人喜好。但是通常来说，attrgetter()要更快一些，而且具有允许同时提取多个字段值的能力。这和针对字典的operator.itemgetter()的使用很类似（参见1.13节）。例如，如果User实例还有一个first_name和last_name属性的话，可以执行如下的排序操作：

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'
```

同样值得一提的是，本节所用到的技术也适用于像min()和max()这样的函数。例如：

```
>>> min(users, key=attrgetter('user_id'))
User(3)
>>> max(users, key=attrgetter('user_id'))
User(99)
>>>
```

1.15　根据字段将记录分组

1.15.1　问题

有一系列的字典或对象实例，我们想根据某个特定的字段（比如说日期）来分组迭代数据。

1.15.2　解决方案

itertools.groupby()函数在对数据进行分组时特别有用。为了说明其用途，假设有如下的字典列表：

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

现在假设想根据日期以分组的方式迭代数据。要做到这些，首先以目标字段（在这个例子中是date）来对序列排序，然后再使用

itertools.groupby()。

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first



rows.sort(key=itemgetter('date'))

# Iterate in groups



for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

这会产生如下的输出：

```
07/01/2012
     {'date': '07/01/2012', 'address': '5412 N CLARK'}
     {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
     {'date': '07/02/2012', 'address': '5800 E 58TH'}
     {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
     {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
     {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
     {'date': '07/04/2012', 'address': '5148 N CLARK'}
     {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

1.15.3 讨论

函数groupby()通过扫描序列找出拥有相同值（或是由参数key指定的函数所返回的值）的序列项，并将它们分组。groupby()创建了一个迭代器，而在每次迭代时都会返回一个值（value）和一个子迭代器（sub_iterator），这个子迭代器可以产生所有在该分组内具有该值的项。

在这里重要的是首先要根据感兴趣的字段对数据进行排序。因为groupby()只能检查连续的项，不首先排序的话，将无法按所想的方式来对记录分组。

如果只是简单地根据日期将数据分组到一起，放进一个大的数据结构中以允许进行随机访问，那么利用defaultdict()构建一个一键多值字典（multidict，见1.6节）可能会更好。例如：

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

这使得我们可以方便地访问每个日期的记录，如下所示：

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

对于后面这个例子，我们并不需要先对记录做排序。因此，如果不考虑内存方面的因素，这种方式会比先排序再用groupby()迭代要来的更快。

1.16 筛选序列中的元素

1.16.1 问题

序列中含有一些数据，我们需要提取出其中的值或根据某些标准对序列做删减。

1.16.2 解决方案

要筛选序列中的数据，通常最简单的方法是使用列表推导式（list comprehension）。例如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for


n in


mylist if


n > 0]
[1, 4, 10, 2, 3]
>>> [n for


n in


mylist if
```

```
n < 0]
[-5, -7, -1]
>>>
```

使用列表推导式的一个潜在缺点是如果原始输入非常大的话，这么做可能会产生一个庞大的结果。如果这是你需要考虑的问题，那么可以使用生成器表达式通过迭代的方式产生筛选的结果。例如：

```
>>> pos = (n for


n in


mylist if


n > 0)
>>> pos
 at 0x1006a0eb0>
>>> for


x in


pos:
...      print


(x)
...
```

```
1
4
10
2
3
>>>
```

有时候筛选的标准没法简单地表示在列表推导式或生成器表达式中。比如，假设筛选过程涉及异常处理或者其他一些复杂的细节。基于此，可以将处理筛选逻辑的代码放到单独的函数中，然后使用内建的filter()函数处理。示例如下：

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def


is_int(val):
    try


:
        x = int(val)
        return


True
    except ValueError


:
        return


False
```

```
ivals = list(filter(is_int, values))
print


(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

filter()创建了一个迭代器，因此如果我们想要的是列表形式的结果，请确保加上了list()，就像示例中那样。

1.16.3　讨论

列表推导式和生成器表达式通常是用来筛选数据的最简单和最直接的方式。此外，它们也具有同时对数据做转换的能力。例如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math



>>> [math.sqrt(n) for


n in


mylist if


n > 0]
```

```
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688
>>>
```

关于筛选数据，有一种情况是用新值替换掉不满足标准的值，而不是丢弃它们。例如，除了要找到正整数之外，我们也许还希望在指定的范围内将不满足要求的值替换掉。通常，这可以通过将筛选条件移到一个条件表达式中来轻松实现。就像下面这样：

```
>>> clip_neg = [n if


n > 0 else


0 for


n in


mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if


n < 0 else


0 for


n in

```

```
mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

另一个值得一提的筛选工具是itertools.compress()，它接受一个可迭代对象以及一个布尔选择器序列作为输入。输出时，它会给出所有在相应的布尔选择器中为True的可迭代对象元素。如果想把对一个序列的筛选结果施加到另一个相关的序列上时，这就会非常有用。例如，假设有以下两列数据：

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK'
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

现在我们想构建一个地址列表，其中相应的count值要大于5。下面是我们可以尝试的方法：

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

这里的关键在于首先创建一个布尔序列，用来表示哪个元素可满足我们的条件。然后compress()函数挑选出满足布尔值为True的相应元素。

同filter()函数一样，正常情况下compress()会返回一个迭代器。因此，如果需要的话，得使用list()将结果转为列表。

1.17　从字典中提取子集

1.17.1　问题

我们想创建一个字典，其本身是另一个字典的子集。

1.17.2　解决方案

利用字典推导式（dictionary comprehension）可轻松解决。例如：

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
   'IBM': 205.55,
   'HPQ': 37.20,
   'FB': 10.75
}

# Make a dictionary of all prices over 200



p1 = { key:value for


key, value in


prices.items() if
```

```
value > 200 }

# Make a dictionary of tech stocks



tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for


key,value in


prices.items() if


key in


tech_names }
```

1.17.3 讨论

大部分可以用字典推导式解决的问题也可以通过创建元组序列然后将它们传给dict()函数来完成。例如：

```
p1 = dict((key, value) for


key, value in


prices.items() if
```

```
value > 200)
```

但是字典推导式的方案更加清晰，而且实际运行起来也要快很多（以本例中的字典prices来测试，效率要高2倍多）。

有时候会有多种方法来完成同一件事情。例如，第二个例子还可以重写成：

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for


key in


prices.keys() & tech_names }
```

但是，计时测试表明这种解决方案几乎要比第一种慢上1.6倍。如果需要考虑性能因素，那么通常都需要花一点时间来研究它。有关计时和性能分析方面的信息，请参见14.13节。

1.18　将名称映射到序列的元素中

1.18.1　问题

我们的代码是通过位置（即索引，或下标）来访问列表或元组的，但有时候这会使代码变得有些难以阅读。我们希望可以通过名称来访问元素，以此减少结构中对位置的依赖性。

1.18.2　解决方案

相比普通的元组，collections.namedtuple()（命名元组）只增加了极小的开销就提供了这些便利。实际上collections.namedtuple()是一个工厂方法，它返回的是Python中标准元组类型的子类。我们提供给它一个类型名称以及相应的字段，它就返回一个可实例化的类、为你已经定义好的字段传入值等。例如：

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
```

```
>>>
```

尽管namedtuple的实例看起来就像一个普通的类实例，但它的实例与普通的元组是可互换的，而且支持所有普通元组所支持的操作，例如索引（indexing）和分解（unpacking）。比如：

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

命名元组的主要作用在于将代码同它所控制的元素位置间解耦。所以，如果从数据库调用中得到一个大型的元组列表，而且通过元素的位置来访问数据，那么假如在表单中新增了一列数据，那么代码就会崩溃。但如果首先将返回的元组转型为命名元组，就不会出现问题。

为了说明这个问题，下面有一些使用普通元组的代码：

```
def
```

```
compute_cost(records):
    total = 0.0
    for


rec in


records:
        total += rec[1] * rec[2]
    return


total
```

通过位置来引用元素常常使得代码的表达力不够强，而且也很依赖于记录的具体结构。下面是使用命名元组的版本：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

当然，如果示例中的records序列已经包含了这样的实例，那么可以避免显式地将记录转换为Stock

命名元组 [4] 。

1.18.3　讨论

namedtuple的一种可能用法是作为字典的替代，后者需要更多的空间来存储。因此，如果要构建涉及字典的大型数据结构，使用namedtuple会更加高效。但是请注意，与字典不同的是，namedtuple是不可变的（immutable）。例如：

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in


<module>
AttributeError


: can't set attribute
>>>
```

如果需要修改任何属性，可以通过使用namedtuple实例的_replace()方法来实现。该方法会创建一个全新的命名元组，并对相应的值做替换。示例如下：

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

_replace()方法有一个微妙的用途，那就是它可以作为一种简便的方法填充具有可选或缺失字段的命名元组。要做到这点，首先创建一个包含默认值的“原型”元组，然后使用_replace()方法创建一个新的实例，把相应的值替换掉。示例如下：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date',

# Create a prototype instance



stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock



def dict_to_stock(s):
return stock_prototype._replace(**s)
```

让我们演示一下上面的代码是如何工作的：

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
```

```
Stock(name='ACME', shares=100, price=123.45, date=None, time=None
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date':
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', t
>>>
```

最后，也是相当重要的是，应该要注意如果我们的目标是定义一个高效的数据结构，而且将来会修改各种实例属性，那么使用namedtuple并不是最佳选择。相反，可以考虑定义一个使用__slots__属性的类（参见8.4节）。

1.19 同时对数据做转换和换算

1.19.1 问题

我们需要调用一个换算（reduction）函数（例如sum()、min()、max()），但首先得对数据做转换或筛选。

1.19.2 解决方案

有一种非常优雅的方式能将数据换算和转换结合在一起——在函数参数中使用生成器表达式。例如，如果想计算平方和，可以像下面这样做：

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for


x in


nums)
```

这里还有一些其他的例子：

```
# Determine if any .py files exist in a directory
```

```
import os



files = os.listdir('dirname')
if


any(name.endswith('.py') for


name in


files):
    print


('There be python!')
else


:
    print


('Sorry, no python.')

# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print


(','.join(str(x) for


x in


s))

# Data reduction across fields of a data structure
portfolio = [
   {'name':'GOOG', 'shares': 50},
   {'name':'YHOO', 'shares': 75},
```

```
    {'name':'AOL', 'shares': 20},
    {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for


s in


portfolio)
```

1.19.3 讨论

这种解决方案展示了当把生成器表达式作为函数的单独参数时在语法上的一些微妙之处（即，不必重复使用括号）。比如，下面这两行代码表示的是同一个意思：

```
s = sum((x * x for


x in


nums))    # Pass generator-expr as argument
s = sum(x * x for


x in


nums)     # More elegant syntax
```

比起首先创建一个临时的列表，使用生成器做参数通常是更为高效和优雅的方式。例如，如果不使用生成器表达式，可能会考虑下面这种实现：

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for


x in


nums])
```

这也能工作，但这引入了一个额外的步骤而且创建了额外的列表。对于这么小的一个列表，这根本就无关紧要，但是如果nums非常巨大，那么就会创建一个庞大的临时数据结构，而且只用一次就要丢弃。基于生成器的解决方案可以以迭代的方式转换数据，因此在内存使用上要高效得多。

某些特定的换算函数比如min()和max()都可接受一个key参数，当可能倾向于使用生成器时会很有帮助。例如在portfolio的例子中，也许会考虑下面这种替代方案：

```
# Original: Returns 20
min_shares = min(s['shares'] for
```

```
s in


portfolio)

# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda


s: s['shares'])
```

1.20　将多个映射合并为单个映射

1.20.1　问题

我们有多个字典或映射，想在逻辑上将它们合并为一个单独的映射结构，以此执行某些特定的操作，比如查找值或检查键是否存在。

1.20.2　解决方案

假设有两个字典：

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

现在假设想执行查找操作，我们必须得检查这两个字典（例如，先在a中查找，如果没找到再去b中查找）。一种简单的方法是利用collections模块中的ChainMap类来解决这个问题。例如：

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x']) # Outputs 1 (from a)
```

```
print(c['y']) # Outputs 2 (from b)

print(c['z']) # Outputs 3 (from a)
```

1.20.3 讨论

ChainMap可接受多个映射然后在逻辑上使它们表现为一个单独的映射结构。但是，这些映射在字面上并不会合并在一起。相反，ChainMap只是简单地维护一个记录底层映射关系的列表，然后重定义常见的字典操作来扫描这个列表。大部分的操作都能正常工作。例如：

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

如果有重复的键，那么这里会采用第一个映射中所对应的值。因此，例子中的c[‘z’]总是引用字典a中的值，而不是字典b中的值。

修改映射的操作总是会作用在列出的第一个映射结构上。例如：

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del


c['x']
>>> a
{'w': 40, 'z': 10}
>>> del


c['y']
Traceback (most recent call last):
...



KeyError: "Key not found in the first mapping: 'y'"
>>>
```

ChainMap与带有作用域的值，比如编程语言中的变量（即全局变量、局部变量等）一起工作时特别有用。实际上这里有一些方法使这个过程变得简单：

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping



>>> values = values.new_child()
```

```
>>> values['x'] = 2
>>> # Add a new mapping



>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping



>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping



>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>
```

作为ChainMap的替代方案，我们可能会考虑利用字典的update()方法将多个字典合并在一起。例如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
```

```
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

这么做行得通，但这需要单独构建一个完整的字典对象（或者修改其中现有的一个字典，这就破坏了原始数据）。此外，如果其中任何一个原始字典做了修改，这个改变都不会反应到合并后的字典中。例如：

```
>>> a['x'] = 13
>>> merged['x']
1
```

而ChainMap使用的就是原始的字典，因此它不会产生这种令人不悦的行为。示例如下：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x']    # Notice change to merged dicts



42
>>>
```

[1]　如果一个对象是可哈希的，那么在它的生存期内必须是不可变的，它需要有一个__hash__()方法。整数、浮点数、字符串、元组都是不可变的。——译者注

[2]　集合的特点就是集合中的元素都是唯一的，但不保证它们之间的顺序。——译者注

[3]　平面文件（flat file）是一种包含没有相对关系结构的记录文件。——译者注

[4]　作者的意思是如果records中的元素是某个类的实例，且已经有了shares和price这样的属性，那就可以直接通过属性名来访问，不需要通过位置来引用，也就没有必要再转换成命名元组了。——译者注

第2章　字符串和文本

无论是解析数据还是产生输出，几乎每一个有实用价值的程序都会涉及某种形式的文本处理。本章的重点放在有关文本操作的常见问题上，例如拆分字符串、搜索、替换、词法分析以及解析。许多任务都可以通过内建的字符串方法轻松解决。但是，更复杂的操作可能会需要用到正则表达式或者创建完整的解析器才能得到解决。以上所有主题本章都有涵盖。此外，本章还提到了一些同Unicode打交道时用到的技巧。

2.1　针对任意多的分隔符拆分字符串

2.1.1　问题

我们需要将字符串拆分为不同的字段，但是分隔符（以及分隔符之间的空格）在整个字符串中并不一致。

2.1.2　解决方案

字符串对象的split()方法只能处理非常简单的情况，而且不支持多个分隔符，对分隔符周围可能存在的空格也无能为力。当需要一些更为灵活的功能时，应该使用re.split()方法：

```
>>> line = 'asdf fjdk; afed, fjek,asdf,     foo'
>>> import re



>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

2.1.3　讨论

re.split()是很有用的，因为可以为分隔符指定多个模式。例如，在上面的解决方案中，分隔符可以是逗号、分号或者是空格符（后面可跟着任意数量的额外空格）。只要找到了对应的模式，无论匹配点的两端是什么字段，整个匹配的结果就成为那个分隔符。最终得到的结果是字段列表，同str.split()得到的结果一样。

当使用re.split()时，需要小心正则表达式模式中的捕获组（capture group）是否包含在了括号中。如果用到了捕获组，那么匹配的文本也会包含在最终结果中。比如，看看下面的结果：

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',',
>>>
```

在特定的上下文中获取到分隔字符也可能是有用的。例如，也许稍后要用到分隔字符来改进字符串的输出：

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']
>>> # Reform the line using the same delimiters
```

```
>>> ''.join(v+d for

v,d in

zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

如果不想在结果中看到分隔字符，但仍然想用括号来对正则表达式模式进行分组，请确保用的是非捕获组，以(?:...)的形式指定。示例如下：

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf','fjdk','afed','fjek','asdf','foo']
>>>
```

2.2 在字符串的开头或结尾处做文本匹配

2.2.1 问题

我们需要在字符串的开头或结尾处按照指定的文本模式做检查，例如检查文件的扩展名、URL协议类型等。

2.2.2 解决方案

有一种简单的方法可用来检查字符串的开头或结尾，只要使用str.startswith()和str.endswith()方法就可以了。示例如下：

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

如果需要同时针对多个选项做检查，只需给

startswith()和endswith()提供包含可能选项的元组即可：

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h')) ]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

这里有另一个例子：

```
from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
             return f.read()
```

奇怪的是，这是Python中需要把元组当成输入的一个地方。如果我们刚好把选项指定在了列表或集合中，请确保首先用tuple()将它们转换成元组。示例如下：

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, no
>>>


url.startswith(tuple(choices))
True
>>>
```

2.2.3 讨论

startswith()和endswith()方法提供了一种非常方便的方式来对字符串的前缀和后缀做基本的检查。类似的操作也可以用切片来完成，但是那种方案不够优雅。例如：

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or


url[:6] == 'https:' or


url[:4] == 'ftp:'
True
>>>
```

可能我们也比较倾向于使用正则表达式作为替

代方案。例如：

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

这也行得通，但是通常对于简单的匹配来说有些过于重量级了。使用本节提到的技术会更简单，运行得也更快。

最后但同样重要的是，当startswith()和endswith()方法和其他操作（比如常见的数据整理操作）结合起来时效果也很好。例如，下面的语句检查目录中有无出现特定的文件：

```
if

any(name.endswith(('.c', '.h')) for

name in

listdir(dirname)):
...
```

2.3　利用**Shell**通配符做字符串匹配

2.3.1　问题

当工作在UNIX Shell下时，我们想使用常见的通配符模式（即，*.py、Dat[0-9]*.csv等）来对文本做匹配。

2.3.2　解决方案

fnmatch模块提供了两个函数——fnmatch()和fnmatchcase()——可用来执行这样的匹配。使用起来很简单：

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
 True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for


name in


names if


fnmatch(name, 'Dat*.csv')]
```

```
['Dat1.csv', 'Dat2.csv']
>>>
```

一般来说，fnmatch()的匹配模式所采用的大小写区分规则和底层文件系统相同（根据操作系统的不同而有所不同）。例如：

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False

>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

如果这个区别对我们而言很重要，就应该使用fnmatchcase()。它完全根据我们提供的大小写方式来匹配：

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

关于这些函数，一个常被忽略的特性是它们在处理非文件名式的字符串时的潜在用途。例如，假设有一组街道地址，就像这样：

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

可以像下面这样写列表推导式：

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9
['5412 N CLARK ST']
>>>
```

2.3.3 讨论

fnmatch所完成的匹配操作有点介乎于简单的字符串方法和全功能的正则表达式之间。如果只是试着在处理数据时提供一种简单的机制以允许使用通配符，那么通常这都是个合理的解决方案。

如果实际上是想编写匹配文件名的代码，那应该使用glob模块来完成，请参见5.13节。

2.4 文本模式的匹配和查找

2.4.1 问题

我们想要按照特定的文本模式进行匹配或查找。

2.4.2 解决方案

如果想要匹配的只是简单的文字，那么通常只需要用基本的字符串方法就可以了，比如str.find()、str.endswith()、str.startswith()或类似的函数。示例如下：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> # Exact match



>>> text == 'yeah'
False

>>> # Match at start or end



>>> text.startswith('yeah')
True
>>> text.endswith('no')
False
```

```
>>> # Search for the location of the first occurrence



>>> text.find('no')
10
>>>
```

对于更为复杂的匹配则需要使用正则表达式以及re模块。为了说明使用正则表达式的基本流程，假设我们想匹配以数字形式构成的日期，比如“11/27/2012”。示例如下：

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re



>>> # Simple matching: \d+ means match one or more digits



>>> if


re.match(r'\d+/\d+/\d+', text1):
...     print


('yes')
... else


:
...     print
```

```
('no')
...



yes
>>> if


re.match(r'\d+/\d+/\d+', text2):
...      print


('yes')
... else


:
... print


('no')
...



no
>>>
```

如果打算针对同一种模式做多次匹配，那么通常会先将正则表达式模式预编译成一个模式对象。例如：

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if


datepat.match(text1):
```

```
...     print

('yes')
... else

:
...     print('no')
...

yes
>>> if

datepat.match(text2):
...     print

('yes')
... else

:
...     print

('no')
...

no
>>>
```

match()方法总是尝试在字符串的开头找到匹配项。如果想针对整个文本搜索出所有的匹配项，那么就应该使用findall()方法。例如：

```
 >>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

当定义正则表达式时，我们常会将部分模式用括号包起来的方式引入捕获组。例如：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

捕获组通常能简化后续对匹配文本的处理，因为每个组的内容都可以单独提取出来。例如：

```
>>> m = datepat.match('11/27/2012')
>>> m

<_sre.SRE_Match object at 0x1005d2750>
>>> # Extract the contents of each group



>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>
```

```
>>> # Find all matches (notice splitting into tuples)



>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for


month, day, year in


datepat.findall(text):
...      print


('{}-{}-{}'.format(year, month, day))
...




2012-11-27
2013-3-13
>>>
```

findall()方法搜索整个文本并找出所有的匹配项然后将它们以列表的形式返回。如果想以迭代的方式找出匹配项，可以使用finditer()方法。示例如下：

```
>>> for


m in
```

```
datepat.finditer(text):
...      print


(m.groups())
...




('11', '27', '2012')
('3', '13', '2013')
>>>
```

2.4.3 讨论

有关正则表达式的基本理论教学超出了本书的范围。但是，本节向您展示了利用re模块来对文本做匹配和搜索的基础。基本功能是首先用re.compile()对模式进行编译，然后使用像match()、findall()或finditer()这样的方法做匹配和搜索。

当指定模式时我们通常会使用原始字符串，比如r'(\d+)/(\d+)/(\d+)'。这样的字符串不会对反斜线字符转义，这在正则表达式上下文中会很有用。否则，我们需要用双反斜线来表示一个单独的'\'，例如'(\d+)/(\d+)/(\d+)'。

请注意match()方法只会检查字符串的开头。有可能出现匹配的结果并不是你想要的情况。例如：

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

如果想要精确匹配，请确保在模式中包含一个结束标记（$），示例如下：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

最后，如果只是想执行简单的文本匹配和搜索操作，通常可以省略编译步骤，直接使用re模块中的函数即可。例如：

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

请注意，如果打算执行很多匹配或查找操作的话，通常需要先将模式编译然后再重复使用。模块级的函数会对最近编译过的模式做缓存处理，因此这里并不会有巨大的性能差异。但是使用自己编译

过的模式会省下一些查找步骤和额外的处理。

2.5 查找和替换文本

2.5.1 问题

我们想对字符串中的文本做查找和替换。

2.5.2 解决方案

对于简单的文本模式，使用str.replace()即可。例如：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

针对更为复杂的模式，可以使用re模块中的sub()函数/方法。为了说明如何使用，假设我们想把日期格式从“11/27/2012”改写为“2012-11-27”。示例如下：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
```

```
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

sub()的第1个参数是要匹配的模式，第2个参数是要替换上的模式。类似“\3”这样的反斜线加数字的符号代表着模式中捕获组的数量。

如果打算用相同的模式执行重复替换，可以考虑先将模式编译以获得更好的性能。示例如下：

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

对于更加复杂的情况，可以指定一个替换回调函数。示例如下：

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

替换回调函数的输入参数是一个匹配对象，由match()或find()返回。用.group()方法来提取匹配中特定的部分。这个函数应该返回替换后的文本。

除了得到替换后的文本外，如果还想知道一共完成了多少次替换，可以使用re.subn()。例如：

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

2.5.3　讨论

除了以上展示的sub()调用之外，关于正则表达式的查找和替换并没有什么更多可说的了。最有技巧性的地方在于指定正则表达式模式——这个最好还是留给读者自己去练习吧。

2.6　以不区分大小写的方式对文本做查找和替换

2.6.1　问题

我们需要以不区分大小写的方式在文本中进行查找，可能还需要做替换。

2.6.2　解决方案

要进行不区分大小写的文本操作，我们需要使用re模块并且对各种操作都要加上re.IGNORECASE标记。例如：

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

上面这个例子揭示出了一种局限，那就是待替换的文本与匹配的文本大小写并不吻合。如果想修正这个问题，需要用到一个支撑函数（support function），示例如下：

```
def

matchcase(word):
    def

replace(m):
         text = m.group()
         if

text.isupper():
             return

word.upper()
         elif

text.islower():
              return

word.lower()
         elif

text[0].isupper():
              return

word.capitalize()
         else

:
              return

word
return
```

```
replace
```

下面是使用这个函数的例子：

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECAS
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

2.6.3 讨论

对于简单的情况，只需加上re.IGNORECASE标记就足以进行不区分大小写的匹配操作了。但请注意的是这对于某些涉及大写转换（case folding）的Unicode匹配来说可能是不够的。具体细节请参见2.10节。

2.7 定义实现最短匹配的正则表达式

2.7.1 问题

我们正在尝试用正则表达式对文本模式做匹配，但识别出来的是最长的可能匹配。相反，我们想将其修改为找出最短的可能匹配。

2.7.2 解决方案

这个问题通常会在匹配的文本被一对开始和结束分隔符包起来的时候出现（例如带引号的字符串）。为了说明这个问题，请看下面的例子：

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

在这个例子中，模式r'\”(.*)\”'尝试去匹配包含在引号中的文本。但是，*操作符在正则表达式中

采用的是贪心策略，所以匹配过程是基于找出最长的可能匹配来进行的。因此，在text2的例子中，它错误地匹配成2个被引号包围的字符串。

要解决这个问题，只要在模式中的*操作符后加上？修饰符就可以了。示例如下：

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

这么做使得匹配过程不会以贪心方式进行，也就会产生出最短的匹配了。

2.7.3 讨论

本节提到了一个当编写含有句点（.）字符的正则表达式时常会遇到的问题。在模式中，句点除了换行符之外可匹配任意字符。但是，如果以开始和结束文本（比如说引号）将句点括起来的话，在匹配过程中将尝试找出最长的可能匹配结果。这会导致匹配时跳过多个开始或结束文本，而将它们都包含在最长的匹配中。在*或+后添加一个？，会强制将匹配算法调整为寻找最短的可能匹配。

2.8 编写多行模式的正则表达式

2.8.1 问题

我们打算用正则表达式对一段文本块做匹配，但是希望在进行匹配时能够跨越多行。

2.8.2 解决方案

这个问题一般出现在希望使用句点（.）来匹配任意字符，但是忘记了句点并不能匹配换行符时。例如，假设想匹配C语言风格的注释：

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
...


multiline comment */
...


'''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

要解决这个问题，可以添加对换行符的支持。示例如下：

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n       multiline comment ']
>>>
```

在这个模式中，(?:.|\n)指定了一个非捕获组（即，这个组只做匹配但不捕获结果，也不会分配组号）。

2.8.3　讨论

re.compile()函数可接受一个有用的标记——re.DOTALL。这使得正则表达式中的句点(.)可以匹配所有的字符，也包括换行符。例如：

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n          multiline comment ']
```

对于简单的情况，使用re.DOTALL标记就可以很好地完成工作。但是如果要处理极其复杂的模式，或者面对的是如2.18节中所描述的为了做分词（tokenizing）而将单独的正则表达式合并在一起的

情况，如果可以选择的话，通常更好的方法是定义自己的正则表达式模式，这样它无需额外的标记也能正确工作。

2.9　将Unicode文本统一表示为规范形式

2.9.1　问题

我们正在同Unicode字符串打交道，但需要确保所有的字符串都拥有相同的底层　表示。

2.9.2　解决方案

在Unicode中，有些特定的字符可以被表示成多种合法的代码点序列。为了说明这个问题，请看下面的示例：

```
>>> s1 = 'Spicy Jalape\u00f1


o'
>>> s2 = 'Spicy Jalapen\u0303


o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
```

```
15
>>>
```

这里的文本“Spicy Jalapeño”以两种形式呈现。第一种使用的是字符“ñ”的全组成（fully composed）形式（U+00F1）。第二种使用的是拉丁字母“n”紧跟着一个“~”组合而成的字符（U+0303）。

对于一个比较字符串的程序来说，同一个文本拥有多种不同的表示形式是个大问题。为了解决这个问题，应该先将文本统一表示为规范形式，这可以通过unicodedata模块来完成：

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'

>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

normalize()的第一个参数指定了字符串应该如

何完成规范表示。NFC表示字符应该是全组成的（即，如果可能的话就使用单个代码点）。NFD表示应该使用组合字符，每个字符应该是能完全分解开的。

Python还支持NFKC和NFKD的规范表示形式，它们为处理特定类型的字符增加了额外的兼容功能。例如：

```
>>> s = '\ufb01


' # A single character
>>> s
'ﬁ'
>>> unicodedata.normalize('NFD', s)
'ﬁ'

# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

2.9.3 讨论

对于任何需要确保以规范和一致性的方式处理Unicode文本的程序来说，规范化都是重要的一部分。尤其是在处理用户输入时接收到的字符串时，此时你无法控制字符串的编码形式，那么规范化文

本的表示就显得更为重要了。

在对文本进行过滤和净化时，规范化同样也占据了重要的部分。例如，假设想从某些文本中去除所有的音符标记（可能是为了进行搜索或匹配）：

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for


c in


t1 if not


unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

最后一个例子展示了unicodedata模块的另一个重要功能——用来检测字符是否属于某个字符类别。使用工具combining()函数可对字符做检查，判断它是否为一个组合型字符。这个模块中还有一些函数可用来查找字符类别、检测数字字符等。

很显然，Unicode是一个庞大的主题。要获得更多有关规范化文本方面的参考信息，可访问 http://www.unicode.org/faq/normalization.html 。Ned Batchelder也在他的网站

http://nedbatchelder.com/text/unipain.html 上对Python中的Unicode处理给出了优秀的示例说明。

2.10 用正则表达式处理Unicode字符

2.10.1 问题

我们正在用正则表达式处理文本，但是需要考虑处理Unicode字符。

2.10.2 解决方案

默认情况下re模块已经对某些Unicode字符类型有了基本的认识。例如，\d已经可以匹配任意Unicode数字字符了：

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII digits



>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>

>>> # Arabic digits



>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

如果需要在模式字符串中包含指定的Unicode字符，可以针对Unicode字符使用转义序列（例如\uFFFF或\UFFFFFFF）。比如，这里有一个正则表达式能在多个不同的阿拉伯代码页中匹配所有的字符：

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08f
>>>
```

当执行匹配和搜索操作时，一个好主意是首先将所有的文本都统一表示为标准形式（见2.9节）。但是，同样重要的是需要注意一些特殊情况。例如，当不区分大小写的匹配和大写转换（case folding）匹配联合起来时，考虑会出现什么行为：

```
>>> pat = re.compile('stra\u00df


e', re.IGNORECASE)
>>> s = 'stra遝'
>>> pat.match(s)                 # Matches



<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper())       # Doesn't match



>>> s.upper()                    # Case folds
```

```
'STRASSE'
>>>
```

2.10.3 讨论

把Unicode和正则表达式混在一起使用绝对是个能让人头痛欲裂的办法。如果真的要这么做，应该考虑安装第三方的正则表达式库（http://pypi.python.org/pypi/regex ），这些第三方库针对Unicode大写转换提供了完整的支持，还包含其他各种有趣的特性，包括近似匹配。

2.11 从字符串中去掉不需要的字符

2.11.1 问题

我们想在字符串的开始、结尾或中间去掉不需要的字符，比如说空格符。

2.11.2 解决方案

strip()方法可用来从字符串的开始和结尾处去掉字符。lstrip()和rstrip()可分别从左或从右侧开始执行去除字符的操作。默认情况下这些方法去除的是空格符，但也可以指定其他的字符。例如：

```
>>> # Whitespace stripping
>>> s = ' hello world \n


'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'
>>>

>>> # Character stripping
>>> t = '-----hello====='
```

```
>>> t.lstrip('-')
'hello====='
>>> t.strip('-=')
'hello'
>>>
```

2.11.3　讨论

当我们读取并整理数据以待稍后的处理时常常会用到这类strip()方法。例如，可以用它们来去掉空格、移除引号等。

需要注意的是，去除字符的操作并不会对位于字符串中间的任何文本起作用。例如：

```
>>> s = ' hello    world     \n


'
>>> s = s.strip()
>>> s
'hello       world'
>>>
```

如果要对里面的空格执行某些操作，应该使用其他技巧，比如使用replace()方法或正则表达式替换。例如：

```
>>> s.replace(' ', '')
'helloworld'
```

```
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

我们通常会遇到的情况是将去除字符的操作同某些迭代操作结合起来，比如说从文件中读取文本行。如果是这样的话，那就到了生成器表达式大显身手的时候了。例如：

```
with open(filename) as f:
lines = (line.strip() for line in f)
for line in lines:
       ...
```

这里，表达式lines = (line.strip() for line in f)的作用是完成数据的转换 [1] 。它很高效，因为这里并没有先将数据读取到任何形式的临时列表中。它只是创建一个迭代器，在所有产生出的文本行上都会执行strip操作。

对于更高级的strip操作，应该转而使用translate()方法。请参见下一节以获得进一步的细节。

2.12　文本过滤和清理

2.12.1　问题

某些无聊的脚本小子在Web页面表单中填入了“pýtĥöñ”这样的文本，我们想以某种方式将其清理掉。

2.12.2　解决方案

文本过滤和清理所涵盖的范围非常广泛，涉及文本解析和数据处理方面的问题。在非常简单的层次上，我们可能会用基本的字符串函数（例如str.upper()和str.lower()）将文本转换为标准形式。简单的替换操作可通过str.replace()或re.sub()来完成，它们把重点放在移除或修改特定的字符序列上。也可以利用unicodedata.normalize()来规范化文本，如2.9节所示。

然而我们可能想更进一步。比方说也许想清除整个范围内的字符，或者去掉音符标志。要完成这些任务，可以使用常被忽视的str.translate()方法。为了说明其用法，假设有如下这段混乱的字符串：

```
>>> s = 'pytĥon\f

is\t

awesome\r\n

'
>>> s
'pytĥon\x0cis\tawesome\r\n'
>>>
```

第一步是清理空格。要做到这步，先建立一个小型的转换表，然后使用translate()方法：

```
>>> remap = {
...

ord('\t

') : ' ',
...

ord('\f

') : ' ',
...

ord('\r

') : None        # Deleted
```

```
...

}
>>> a = s.translate(remap)
>>> a
'pythôn is awesome\n'
>>>
```

可以看到，类似\t和\f这样的空格符已经被重新映射成一个单独的空格。回车符\r已经完全被删除掉了。

可以利用这种重新映射的思想进一步构建出更加庞大的转换表。例如，我们把所有的Unicode组合字符都去掉：

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                              if unicodedata.combining(chr(c
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pythôn is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

在这个例子中，我们使用dict.fromkeys()方法构

建了一个将每个Unicode组合字符都映射为None的字典。

原始输入会通过unicodedata.normalize()方法转换为分离形式，然后再通过translate()方法删除所有的重音符号。我们也可以利用相似的技术来去掉其他类型的字符（例如控制字符）。

下面来看另一个例子。这里有一张转换表将所有的Unicode十进制数字字符映射为它们对应的ASCII版本：

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...                  for


c in


range(sys.maxunicode)
...                  if


unicodedata.category(chr(c)) == 'Nd' }
...



>>> len(digitmap)
460
>>> # Arabic digits



>>> x = '\u0661\u0662\u0663
```

```
'
>>> x.translate(digitmap)
'123'
>>>
```

另一种用来清理文本的技术涉及I/O解码和编码函数。大致思路是首先对文本做初步的清理，然后通过结合encode()和decode()操作来修改或清理文本。示例如下：

```
>>> a
'pythôn is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

这里的normalize()方法先对原始文本做分解操作。后续的ASCII编码/解码只是简单地一次性丢弃所有不需要的字符。很显然，这种方法只有当我们的最终目标就是ASCII形式的文本时才有用。

2.12.3　讨论

文本过滤和清理的一个主要问题就是运行时的性能。一般来说操作越简单，运行得就越快。对于简单的替换操作，用str.replace()通常是最快的方式

——即使必须多次调用它也是如此。比方说如果要清理掉空格符，可以编写如下的代码：

```
def

clean_spaces(s):
    s = s.replace('\r

', '')
    s = s.replace('\t

', ' ')
    s = s.replace('\f

', ' ')
return

s
```

如果试着调用它，就会发现这比使用translate()或者正则表达式的方法要快得多。

另一方面，如果需要做任何高级的操作，比如字符到字符的重映射或删除，那么translate()方法还是非常快的。

从整体来看，我们应该在具体的应用中去进一步揣摩性能方面的问题。不幸的是，想在技术上给

出一条“放之四海而皆准”的建议是不可能的，所以应该尝试多种不同的方法，然后做性能统计分析。

尽管本节的内容主要关注的是文本，但类似的技术也同样适用于字节对象（byte），这包括简单的替换、翻译和正则表达式。

2.13 对齐文本字符串

2.13.1 问题

我们需要以某种对齐方式将文本做格式化处理。

2.13.2 解决方案

对于基本的字符串对齐要求，可以使用字符串的ljust()、rjust()和center()方法。示例如下：

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World '
>>> text.rjust(20)
'         Hello World'
>>> text.center(20)
'    Hello World '
>>>
```

所有这些方法都可接受一个可选的填充字符。例如：

```
>>> text.rjust(20,'=')
'=========Hello World'
>>> text.center(20,'*')
'****Hello World*****'
```

```
>>>
```

format()函数也可以用来轻松完成对齐的任务。需要做的就是合理利用'<'、'>'，或'^'字符以及一个期望的宽度值 [2] 。例如：

```
>>> format(text, '>20')
'         Hello World'
>>> format(text, '<20')
'Hello World '
>>> format(text, '^20')
'     Hello World '
>>>
```

如果想包含空格之外的填充字符，可以在对齐字符之前指定：

```
>>> format(text, '=>20s')
'=========Hello World'
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

当格式化多个值时，这些格式化代码也可以用在format()方法中。例如：

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'    Hello      World'
>>>
```

```
```

format()的好处之一是它并不是特定于字符串的。它能作用于任何值，这使得它更加通用。例如，可以对数字做格式化处理：

```
>>> x = 1.2345
>>> format(x, '>10')
'    1.2345'
>>> format(x, '^10.2f')
'   1.23 '
>>>
```

2.13.3　讨论

在比较老的代码中，通常会发现%操作符用来格式化文本。例如：

```
>>> '%-20s' % text
'Hello World '
>>> '%20s' % text
'         Hello World'
>>>
```

但是在新的代码中，我们应该会更钟情于使用format()函数或方法。format()比%操作符提供的功能要强大多了。此外，format()可作用于任意类型的对象，比字符串的ljust()、rjust()以及center()方法

要更加通用。

想了解format()函数的所有功能，请参考Python的在线手册http://docs.python.org/3/library/ string. html#formatspec 。

2.14　字符串连接及合并

2.14.1　问题

我们想将许多小字符串合并成一个大的字符串。

2.14.2　解决方案

如果想要合并的字符串在一个序列或可迭代对象中，那么将它们合并起来的最快方法就是使用join()方法。示例如下：

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

初看上去语法可能显得有些怪异，但是join()操作其实是字符串对象的一个方法。这么设计的部分原因是因为想要合并在一起的对象可能来自于各种不同的数据序列，比如列表、元组、字典、文件、

集合或生成器，如果单独在每一种序列对象中实现一个join()方法就显得太冗余了。因此只需要指定想要的分隔字符串，然后在字符串对象上使用join()方法将文本片段粘合在一起就可以了。

如果只是想连接一些字符串，一般使用+操作符就足够完成任务了：

```
>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>
```

针对更加复杂的字符串格式化操作，+操作符同样可以作为format()的替代，很好地完成任务：

```
>>> print


('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print


(a + ' ' + b)
Is Chicago Not Chicago?
>>>
```

如果打算在源代码中将字符串字面值合并在一

起，可以简单地将它们排列在一起，中间不加+操作符。示例如下：

```
>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>
```

2.14.3 讨论

字符串连接这个主题可能看起来还没有高级到要用一整节的篇幅来讲解，但是程序员常常会在这个问题上做出错误的编程选择，使得他们的代码性能受到影响。

最重要的一点是要意识到使用+操作符做大量的字符串连接是非常低效的，原因是由于内存拷贝和垃圾收集产生的影响。特别是你绝不会想写出这样的字符串连接代码：

```
s = ''
for


p in


parts:
    s += p
```

这种做法比使用join()方法要慢上许多。主要是因为每个+=操作都会创建一个新的字符串对象。我们最好先收集所有要连接的部分，最后再一次将它们连接起来。

一个相关的技巧（很漂亮的技巧）是利用生成器表达式（见1.19节）在将数据转换为字符串的同时完成连接操作。示例如下：

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for


d in


data)
'ACME,50,91.1'
>>>
```

对于不必要的字符串连接操作也要引起重视。有时候在技术上并非必需的时候，程序员们也会忘乎所以地使用字符串连接操作。例如在打印的时候：

```
print


(a + ':' + b + ':' + c)     # Ugly
```

```
print

(':'.join([a, b, c]))       # Still ugly


print

(a, b, c, sep=':')          # Better
```

将字符串连接同I/O操作混合起来的时候需要对应用做仔细的分析。例如，考虑如下两段代码：

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)
# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

如果这两个字符串都很小，那么第一个版本的代码能带来更好的性能，这是因为执行一次I/O系统调用的固有开销就很高。另一方面，如果这两个字符串都很大，那么第二个版本的代码会更加高效。因为这里避免了创建大的临时结果，也没有对大块的内存进行拷贝。这里必须再次强调，你需要对自己的数据做分析，以此才能判定哪一种方法可

以获得最好的性能。

　　最后但也是最重要的是，如果我们编写的代码要从许多短字符串中构建输出，则应该考虑编写生成器函数，通过yield关键字生成字符串片段。示例如下：

```
def


sample():
    yield


'Is'
    yield


'Chicago'
    yield


'Not'
    yield


'Chicago?'
```

　　关于这种方法有一个有趣的事实，那就是它不会假设产生的片段要如何组合在一起。比如说可以用join()将它们简单的连接起来：

```
text = ''.join(sample())
```

或者，也可以将这些片段重定向到I/O：

```
for

part in

sample():

f.write(part)
```

又或者我们能以混合的方式将I/O操作智能化地结合在一起：

```
def

combine(source, maxsize):
     parts = []
     size = 0
     for

part in

source:
         parts.append(part)
         size += len(part)
         if
```

```
size > maxsize:
             yield

''.join(parts)


parts = []


size = 0
    yield

''.join(parts)

for

part in

combine(sample(), 32768):


f.write(part)
```

关键在于这里的生成器函数并不需要知道精确的细节，它只是产生片段而已。

2.15 给字符串中的变量名做插值处理

2.15.1 问题

我们想创建一个字符串，其中嵌入的变量名称会以变量的字符串值形式替换掉。

2.15.2 解决方案

Python并不直接支持在字符串中对变量做简单的值替换。但是，这个功能可以通过字符串的format()方法近似模拟出来。示例如下：

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

另一种方式是，如果要被替换的值确实能在变量中找到，则可以将format_map()和vars()联合起来使用，示例如下：

```
>>> name = 'Guido'
>>> n = 37
```

```
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

有关vars()的一个微妙的特性是它也能作用于类实例上。比如：

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido',37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

而format()和format_map()的一个缺点则是没法优雅地处理缺少某个值的情况。例如：

```
>>> s.format(name='Guido')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

避免出现这种情况的一种方法就是单独定义一个带有__missing__()方法的字典类，示例如下：

```
class safesub(dict):
```

```
    def __missing__(self, key):
        return '{' + key + '}'
```

现在用这个类来包装传给format_map()的输入参数：

```
>>> del

n     # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

如果发现自己在代码中常常需要执行这些步骤，则可以将替换变量的过程隐藏在一个小型的功能函数内，这里要采用一种称之为“frame hack”的技巧 [3] 。示例如下：

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

现在，我们就可以像这样编写代码了：

```
>>> name = 'Guido'
>>> n = 37
>>> print
```

```
(sub('Hello {name}'))
Hello Guido
>>> print


(sub('You have {n} messages.'))
You have 37 messages.
>>> print


(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

2.15.3 讨论

多年来，由于Python缺乏真正的变量插值功能，由此产生了各种解决方案。作为本节中已给出的解决方案的替代，有时候我们会看到类似下面代码中的字符串格式化操作：

```
>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>
```

我们可能还会看到模板字符串（template string）的使用：

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

但是，format()和format_map()方法比上面这些替代方案都要更加现代化，我们应该将其作为首选。使用format()的一个好处是可以同时得到所有关于字符串格式化方面的功能（对齐、填充、数值格式化等），而这些功能在字符串模板对象上是不可能做到的。

在本节的部分内容中还提到了一些有趣的高级特性。字典类中鲜为人知的__missing__()方法可用来处理缺少值时的行为。在safesub类中，我们将该方法定义为将缺失的值以占位符的形式返回，因此这里不会抛出KeyError异常，缺少的那个值会出现在最后生成的字符串中（可能对调试有些帮助）。

sub()函数使用了sys._getframe(1)来返回调用方的栈帧。通过访问属性f_locals来得到局部变量。无需赘言，在大部分的代码中都应该避免去和栈帧打交道，但是对于类似完成字符串替换功能的函数来说，这会是有用的。插一句题外话，值得指出的是f_locals是一个字典，它完成对调用函数中局部变量的拷贝。尽管可以修改f_locals的内容，可是修改后

并不会产生任何持续性的效果。因此，尽管访问不同的栈帧可能看起来是很邪恶的，但是想意外地覆盖或修改调用方的本地环境也是不可能的。

2.16 以固定的列数重新格式化文本

2.16.1 问题

我们有一些很长的字符串，想将它们重新格式化，使得它们能按照用户指定的列数来显示。

2.16.2 解决方案

可以使用textwrap模块来重新格式化文本的输出。例如，假设有如下这段长字符串：

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \

the eyes, not around the eyes, don't look around the eyes, \

look into my eyes, you're under."
```

这里可以用textwrap模块以多种方式来重新格式化字符串：

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eye
not around the eyes, don't look around the eyes, look into my eye
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, initial_indent=' '))
Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, subsequent_indent=' '))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not
around the eyes, don't look around
the eyes, look into my eyes, you're
under.
```

2.16.3 讨论

textwrap模块能够以简单直接的方式对文本格式做整理使其适合于打印——尤其是当希望输出结果能很好地显示在终端上时。关于终端的尺寸大小，可以通过os.get_ terminal_size()来获取。例如：

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

fill()方法还有一些额外的选项可以用来控制如何处理制表符、句号等。请参阅textwrap.TextWrapper类的文档（http://docs.python.org/3.3/library/ textwrap. html# text w r ap.TextWrapper ）以获得进一步的细节。

2.17　在文本中处理HTML和XML实体

2.17.1　问题

我们想将&entity或&#code这样的HTML或XML实体替换为它们相对应的文本。或者，我们需要生成文本，但是要对特定的字符（比如<,>或&）做转义处理。

2.17.2　解决方案

如果要生成文本，使用html.escape()函数来完成替换<or>这样的特殊字符相对来说是比较容易的。例如：

```
>>> s = 'Elements are written as "<tag>text</tag>".'
>>> import html



>>> print


(s)
Elements are written as "<tag>text</tag>".
>>> print
```

```
(html.escape(s))
Elements are written as "<tag>text</tag>".

>>> # Disable escaping of quotes



>>> print


(html.escape(s, quote=False))
Elements are written as "<tag>text</tag>".
>>>
```

如果要生成ASCII文本，并且想针对非ASCII字符将它们对应的字符编码实体嵌入到文本中，可以在各种同I/O相关的函数中使用errors='xmlcharrefreplace'参数来实现。示例如下：

```
>>> s = 'Spicy Jalapen~o'
>>> s.encode('ascii', errors='xmlcharrefreplace')
b'Spicy Jalape&#241;o'
>>>
```

要替换文本中的实体，那就需要不同的方法。如果实际上是在处理HTML或XML，首先应该尝试使用一个合适的HTML或XML解析器。一般来说，这些工具在解析的过程中会自动处理相关值的替换，而我们完全无需为此操心。

如果由于某种原因在得到的文本中带有一些实

体，而我们想手工将它们替换掉，通常可以利用各种HTML或XML解析器自带的功能函数和方法来完成。示例如下：

```
>>> s = 'Spicy "Jalapeño&quot.'
>>> from html.parser import


HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
'Spicy "Jalapen~o".'
>>>

>>> t = 'The prompt is >>>'
>>> from xml.sax.saxutils import


unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

2.17.3　讨论

在生成HTML或XML文档时，适当地对特殊字符做转义处理常常是个容易被忽视的细节。尤其是当自己用print()或其他一些基本的字符串格式化函数来产生这类输出时更是如此。简单的解决方案是使用像html.escape()这样的工具函数。

如果需要反过来处理文本（即，将HTML或

XML实体转换成对应的字符），有许多像xml.sax.saxutils.unescape()这样的工具函数能帮上忙。但是，我们需要仔细考察一个合适的解析器应该如何使用。例如，如果是处理HTML或XML，像html.parser或xml.etree.ElementTree这样的解析模块应该已经解决了有关替换文本中实体的细节问题。

2.18 文本分词

2.18.1 问题

我们有一个字符串，想从左到右将它解析为标记流（stream of tokens）。

2.18.2 解决方案

假设有如下的字符串文本：

```
text = 'foo = 23 + 42 * 10'
```

要对字符串做分词处理，需要做的不仅仅只是匹配模式。我们还需要有某种方法来识别出模式的类型。例如，我们可能想将字符串转换为如下的序列对：

```
tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+
          ('NUM', '42'), ('TIMES', '*'), ('NUM', 10')]
```

要完成这样的分词处理，第一步是定义出所有可能的标记，包括空格。这可以通过正则表达式中

的命名捕获组来实现，示例如下：

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ     = r'(?P<EQ>=)'
WS     = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]
```

在这些正则表达式模式中，形如?P<TOKENNAME>这样的约定是用来将名称分配给该模式的。这个我们稍后会用到。

接下来我们使用模式对象的scanner()方法来完成分词操作。该方法会创建一个扫描对象，在给定的文本中重复调用match()，一次匹配一个模式。下面这个交互式示例展示了扫描对象是如何工作的：

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
```

```
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

要利用这项技术并将其转化为代码，我们可以做些清理工作然后轻松地将其包含在一个生成器函数中，示例如下：

```
from collections import namedtuple

Token = namedtuple('Token', ['type','value'])
def generate_tokens(pat, text):
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use



for tok in generate_tokens(master_pat, 'foo = 42'):
     print(tok)

# Produces output



# Token(type='NAME', value='foo')



# Token(type='WS', value=' ')
```

```
# Token(type='EQ', value='=')

# Token(type='WS', value=' ')

# Token(type='NUM', value='42')
```

如果想以某种方式对标记流做过滤处理，要么定义更多的生成器函数，要么就用生成器表达式。例如，下面的代码告诉我们如何过滤掉所有的空格标记。

```
tokens = (tok for

tok in

generate_tokens(master_pat, text)
          if

tok.type != 'WS')
for

tok in
```

```
tokens:
    print


(tok)
```

2.18.3 讨论

对于更加高级的文本解析，第一步往往是分词处理。要使用上面展示的扫描技术，有几个重要的细节需要牢记于心。第一，对于每个可能出现在输入文本中的文本序列，都要确保有一个对应的正则表达式模式可以将其识别出来。如果发现有任何不能匹配的文本，扫描过程就会停止。这就是为什么有必要在上面的示例中指定空格标记（WS）。

这些标记在正则表达式（即re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))）中的顺序同样也很重要。当进行匹配时，re模块会按照指定的顺序来对模式做匹配。因此，如果碰巧某个模式是另一个较长模式的子串时，就必须确保较长的那个模式要先做匹配。示例如下：

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
```

```
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```

第2个模式是错误的（注释掉的那一行），因为这样会把文本'<='匹配为LT（'<'）紧跟着EQ（'='），而没有匹配为单独的标记LE（'<='），这与我们的本意不符。

最后也最重要的是，对于有可能形成子串的模式要多加小心。例如，假设有如下两种模式：

```
PRINT = r'(P<PRINT>print)'
NAME  = r'(P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'

master_pat = re.compile('|'.join([PRINT, NAME]))

for tok in generate_tokens(master_pat, 'printer'):
    print(tok)

# Outputs :



# Token(type='PRINT', value='print')



# Token(type='NAME', value='er')
```

对于更加高级的分词处理，我们应该去看看像PyParsing或PLY这样的包。有关PLY的例子将在下一节中讲解。

2.19 编写一个简单的递归下降解析器

2.19.1 问题

我们需要根据一组语法规则来解析文本，以此执行相应的操作或构建一个抽象语法树来表示输入。语法规则很简单，因此我们倾向于自己编写解析器而不是使用某种解析器框架。

2.19.2 解决方案

在这个问题中，我们把重点放在根据特定的语法来解析文本上。要做到这些，应该以BNF或EBNF的形式定义出语法的正式规格。比如，对于简单的算术运算表达式，语法看起来是这样的：

```
expr ::= expr + term
     | expr - term
     | term
term ::= term * factor
     | term / factor
     | factor
factor ::= ( expr )
       | NUM
```

又或者以EBNF的形式定义为如下形式：

```
expr ::= term { (+|-) term }*
term ::= factor { (*|/) factor }*
factor ::= ( expr )
     | NUM
```

在EBNF中，部分包括在{ ... }*中的规则是可选的。*意味着零个或更多重复项（和在正则表达式中的意义相同）。

现在，如果我们对BNF还不熟悉的话，可以把它看做是规则替换或取代的一种规范形式，左侧的符号可以被右侧的符号所取代（反之亦然）。一般来说，在解析的过程中我们会尝试将输入的文本同语法做匹配，通过BNF来完成各种替换和扩展。为了说明，假设正在解析一个类似于3 + 4 * 5这样的表达式。这个表达式首先应该被分解为标记流，这可以使用2.18节中描述的技术来实现。得到的结果可能是下面这样的标记序列：

```
NUM + NUM * NUM
```

从这里开始，解析过程就涉及通过替换的方式将语法匹配到输入标记上：

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor}* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

完成所有的替换需要花上一段时间，这是由输入的规模和尝试去匹配的语法规则所决定的。第一个输入标记是一个NUM，因此替换操作首先会把重点放在匹配这一部分上。一旦匹配上了，重点就转移到下一个标记+上，如此往复。当发现无法匹配下一个标记时，右手侧的特定部分（{ (*/) factor }*）就会消失。在一个成功的解析过程中，整个右手侧部分会完全根据匹配到的输入标记流来相应地扩展。

有了前面这些基础，下面就向各位展示如何构建一个递归下降的表达式计算器：

```
import re
import collections

# Token specification
```

```
NUM    = r'(?P<NUM>\d+)'
PLUS   = r'(?P<PLUS>\+)'
MINUS  = r'(?P<MINUS>-)'
TIMES  = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))
# Tokenizer



Token = collections.namedtuple('Token', ['type','value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Parser



class ExpressionEvaluator:
'''



Implementation of a recursive descent parser. Each method



implements a single grammar rule. Use the ._accept() method



to test and accept the current lookahead token. Use the ._expect
```

```
method to exactly match and discard the next token on on the inpu



(or raise a SyntaxError if it doesn't match).



'''





def parse(self,text):
    self.tokens = generate_tokens(text)
    self.tok = None               # Last symbol consumed



    self.nexttok = None          # Next symbol tokenized



    self._advance()               # Load first lookahead token



    return self.expr()

def _advance(self):
    'Advance one token ahead'
    self.tok, self.nexttok = self.nexttok, next(self.tokens, None

def _accept(self,toktype):
    'Test and consume the next token if it matches toktype'
    if self.nexttok and self.nexttok.type == toktype:
        self._advance()
        return True
    else:
        return False

def _expect(self,toktype):
    'Consume next token if it matches toktype or raise SyntaxErro
     if not self._accept(toktype):
         raise SyntaxError('Expected ' + toktype)
```

```
# Grammar rules follow

def expr(self):
    "expression ::= term { ('+'|'-') term }*"

    exprval = self.term()
    while self._accept('PLUS') or self._accept('MINUS'):
        op = self.tok.type
        right = self.term()
        if op == 'PLUS':
            exprval += right
        elif op == 'MINUS':
            exprval -= right
    return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }*"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval *= right
        elif op == 'DIVIDE':
            termval /= right
    return termval

def factor(self):
    "factor ::= NUM | ( expr )"

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
         raise SyntaxError('Expected NUMBER or LPAREN')
```

下面是以交互式的方式使用ExpressionEvaluator类的示例：

```
>>> e = ExpressionEvaluator()
>>> e.parse('2')
2
>>> e.parse('2 + 3')
5
>>> e.parse('2 + 3 * 4')
14
>>> e.parse('2 + (3 + 4) * 5')
37
>>> e.parse('2 + (3 + * 4)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exprparse.py", line 40, in parse
     return


self.expr()
  File "exprparse.py", line 67, in expr
     right = self.term()
  File "exprparse.py", line 77, in term
     termval = self.factor()
  File "exprparse.py", line 93, in factor
     exprval = self.expr()
  File "exprparse.py", line 67, in expr
     right = self.term()
  File "exprparse.py", line 77, in term
     termval = self.factor()
  File "exprparse.py", line 97, in factor
     raise SyntaxError


("Expected NUMBER or LPAREN")
SyntaxError: Expected NUMBER or LPAREN
>>>
```

如果我们想做的不只是纯粹的计算，那就需要修改ExpressionEvaluator类来实现。比如，下面的实现构建了一棵简单的解析树：

```
class ExpressionTreeBuilder(ExpressionEvaluator):
    def expr(self):
        "expression ::= term { ('+'|'-') term }"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval = ('+', exprval, right)
            elif op == 'MINUS':
                 exprval = ('-', exprval, right)
        return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval = ('*', termval, right)
        elif op == 'DIVIDE':
              termval = ('/', termval, right)
        return termval

def factor(self):
    'factor ::= NUM | ( expr )'

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
         raise SyntaxError('Expected NUMBER or LPAREN')
```

下面的示例展示了它是如何工作的：

```
>>> e = ExpressionTreeBuilder()
>>> e.parse('2 + 3')
('+', 2, 3)
>>> e.parse('2 + 3 * 4')
('+', 2, ('*', 3, 4))
>>> e.parse('2 + (3 + 4) * 5')
('+', 2, ('*', ('+', 3, 4), 5))
>>> e.parse('2 + 3 + 4')
('+', ('+', 2, 3), 4)
>>>
```

2.19.3 讨论

文本解析是一个庞大的主题，一般会占用学生们编译原理课程的前三周时间。如果你正在寻找有关语法、解析算法和其他相关信息的背景知识，那么应该去找一本编译器方面的图书来读。无需赘言，本书是不会重复那些内容的。

然而，要编写一个递归下降的解析器，总体思路还是比较简单的。我们要将每一条语法规则转变为一个函数或方法。因此，如果我们的语法看起来是这样的：

```
expr ::= term { ('+'|'-') term }*
```

```
term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
       | NUM
```

就可以像下面这样将它们转换为对应的方法：

```
class ExpressionEvaluator:
    ...
    def expr(self):
        ...
    def term(self):
        ...

    def factor(self):
        ...
```

每个方法的任务很简单——必须针对语法规则的每个部分从左到右扫描，在扫描过程中处理符号标记。从某种意义上说，这些方法的目的就是顺利地将规则消化掉，如果卡住了就产生一个语法错误。要做到这点，需要应用下面这些实现技术。

- 如果规则中的下一个符号标记是另一个语法规则的名称（例如，term或者factor），就需要调用同名的方法。这就是算法中的“下降”部分——控制其下降到另一个语法规则中。有时候规则中会涉及调用已经在执行的方法（例

如，在规则factor ::= '(' expr ')'中对expr的调用）。这就是算法中的“递归”部分。

- 如果规则中的下一个符号标记是一个特殊的符号（例如'('），需要检查下一个标记，看它们是否能完全匹配。如果不能匹配，这就是语法错误。本节给出的_expect()方法就是用来处理这些步骤的。

- 如果规则中的下一个符号标记存在多种可能的选择（例如+或-），则必须针对每种可能性对下一个标记做检查，只有在有匹配满足时才前进到下一步。这就是本节给出的_accept()方法的目的所在。这有点像_except()的弱化版，在_accept()中如果有匹配满足，就前进到下一步，但如果没有匹配，它只是简单的回退而不会引发一个错误（这样检查才可以继续进行下去）。

- 对于语法规则中出现的重复部分（例如expr ::=term { ('+' | '-') term }*）,这是通过while循环来实现的。一般在循环体中收集或处理所有的重复项，直到无法找到更多的重复项为止。

- 一旦整个语法规则都已经处理完，每个方法就返回一些结果给调用者。这就是在解析过

程中将值进行传递的方法。比如，在计算器表达式中，表达式解析的部分结果会作为值来返回。最终它们会结合在一起，在最顶层的语法规则方法中得到执行。

尽管本节给出的例子很简单，但递归下降解析器可以用来实现相当复杂的解析器。例如，Python代码本身也是通过一个递归下降解析器来解释的。如果对此很感兴趣，可以通过检查Python源代码中的Grammar/Grammar文件来一探究竟。即便如此，要自己手写一个解析器时仍然需要面对各种陷阱和局限。

局限之一就是对于任何涉及左递归形式的语法规则，都没法用递归下降解析器来解决。例如，假设需要解释如下的规则：

```
items ::= items ',' item
      | item
```

要完成这样的解析，我们可能会试着这样来定义items()方法：

```
def

items(self):
```

```
    itemsval = self.items()
    if


itemsval and


self._accept(','):
        itemsval.append(self.item())
    else



:
        itemsval = [ self.item() ]
```

唯一的问题就是这么做行不通。实际上这会产生一个无穷递归的错误。

我们也可能会陷入到语法规则自身的麻烦中。例如，我们可能想知道表达式是否能以这种加简单的语法形式来描述：

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
        | NUM
```

这个语法从技术上说是能实现的，但是它却并没有遵守标准算术中关于计算顺序的约定。比如说，表达式“3 + 4 * 5”会被计算为35，而不是我们预期的23。因此这里需要单独的“expr”和“term”规

则来确保计算结果的正确性。

对于真正复杂的语法解析，最好还是使用像PyParsing或PLY这样的解析工具。如果使用PLY的话，解析计算器表达式的代码看起来是这样的：

```
from ply.lex import lex
from ply.yacc import yacc

# Token list



tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN',

# Ignored characters




t_ignore = ' \t\n'

# Token specifications (as regexs)



t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token processing functions



def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

```
# Error handler

def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer

lexer = lex()

# Grammar rules and handler functions

def p_expr(p):
    '''

    expr : expr PLUS term

    | expr MINUS term

    '''

if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
```

```
    '''

    expr : term

    '''

    p[0] = p[1]

def p_term(p):
    '''

    term : term TIMES factor

         | term DIVIDE factor

    '''

    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    '''

    term : factor
```

```
    '''



    p[0] = p[1]

def p_factor(p):
    '''



    factor : NUM



    '''



    p[0] = p[1]

def p_factor_group(p):
    '''



    factor : LPAREN expr RPAREN



    '''



    p[0] = p[2]

def p_error(p):
    print('Syntax error')

parser = yacc()
```

在这份代码中会发现所有的东西都是以一种更高层的方式来定义的。我们只需编写匹配标记符号的正则表达式，以及当匹配各种语法规则时所需要的高层处理函数就行了。而实际运行解析器、接收符号标记等都完全由库来实现。

下面是如何使用解析器对象的示例：

```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

如果想在编程中增加一点激动兴奋的感觉，编写解析器和编译器会是非常有趣的课题。再次说明，一本编译器方面的教科书会涵盖许多理论之下的底层细节。但是，在网上同样也能找到许多优秀的在线资源。Python自带的ast模块也同样值得去看看。

2.20 在字节串上执行文本操作

2.20.1 问题

我们想在字节串（Byte String）上执行常见的文本操作（例如，拆分、搜索和替换）。

2.20.2 解决方案

字节串已经支持大多数和文本字符串一样的内建操作。例如：

```
>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>
```

类似这样的操作在字节数组上也能完成。例如：

```
>>> data = bytearray(b'Hello World')
>>> data[0:5]
```

```
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>
```

我们可以在字节串上执行正则表达式的模式匹配操作，但是模式本身需要以字节串的形式来指定。示例如下：

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re



>>> re.split('[:,]',data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/re.py", line 191, in split
    return


_compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object

>>> re.split(b'[:,]',data)  # Notice: pattern as bytes



[b'FOO', b'BAR', b'SPAM']
>>>
```

2.20.3 讨论

就绝大部分情况而言，几乎所有能在文本字符串上执行的操作同样也可以在字节串上进行。但是，还是有几个显著的区别值得大家注意。例如：

```
>>> a = 'Hello World'       # Text string



>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World'     # Byte string



>>> b[0]
72
>>> b[1]
101
>>>
```

这种语义上的差异会对试图按照字符的方式处理面向字节流数据的程序带来影响。

其次，字节串并没有提供一个漂亮的字符串表示，因此打印结果并不干净利落，除非首先将其解码为文本字符串。示例如下：

```
>>> s = b'Hello World'
>>> print
```

```
(s)
b'Hello World'                      # Observe b'...'
>>> print


(s.decode('ascii'))
Hello World
>>>
```

同样道理，在字节串上是没有普通字符串那样的格式化操作的。

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple

>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

如果想在字节串上做任何形式的格式化操作，应该使用普通的文本字符串然后再做编码。示例如下：

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('a
b'ACME                  100      490.10'
>>>
```

最后，需要注意的是使用字节串会改变某些特定操作的语义——尤其是那些与文件系统相关的操作。例如，如果提供一个以字节而不是文本字符串来编码的文件名，文件系统通常都会禁止对文件名的编码/解码。示例如下：

```
>>> # Write a UTF-8 filename
>>> with


open('jalape\xf1


o.txt', 'w') as


f:
...


f.write('spicy')
...




>>> # Get a directory listing
>>> import os



>>> os.listdir('.')            # Text string (names are decoded)
['Jalapeño.txt']
>>> os.listdir(b'.')          # Byte string (names left as bytes
[b'jalapen\xcc\x83o.txt']
>>>
```

请注意这个例子中的最后部分，本例中以字节串作为目录名从而导致产生的名称以未经编码的原始字节形式返回。在显示目录内容时，文件名包含了原始的UTF-8编码。有关文件名的处理请参阅5.15节。

最后要说的是，有些程序员可能会因为性能上有可能得到提升而倾向于将字节串作为文本字符串的替代来使用。尽管操纵字节确实要比文本来的略微高效一些（由于同Unicode相关的固有开销较高），但这么做通常会导致非常混乱和不符合语言习惯的代码。我们常会发现字节串和Python中许多其他部分并不能很好地相容，这样为了保证结果的正确性，我们只能手动去执行各种各样的编码/解码操作。坦白地说，如果要同文本打交道，在程序中使用普通的文本字符串就好，不要用字节串。

---

[1]　把原始数据中每一行开头和结尾处的空格符去掉，相当于一种转换处理。——译者注

[2]　'>'表示右对齐，'<'表示左对齐，'^'表示居中对齐，这些字符称为对齐字符。——译者注

[3]　即需要同函数的栈帧打交道。sys._getframe这个特殊的函数可以让我们获得调用函数的栈信息。

——译者注

第3章　数字、日期和时间

在Python中对整数和浮点数进行数学计算是非常容易的。但是，如果需要对分数、数组或者日期和时间进行计算，就需要完成更多的工作。本章的重点正是应对这些主题。

3.1　对数值进行取整

3.1.1　问题

我们想将一个浮点数取整到固定的小数位。

3.1.2　解决方案

对于简单的取整操作，使用内建的round(value, ndigits)函数即可。示例如下：

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361,3)
1.254
>>>
```

当某个值恰好等于两个整数间的一半时，取整操作会取到离该值最接近的那个偶数上。也就是说，像1.5或2.5这样的值都会取整到2。

传递给round()的参数ndigits可以是负数，在这种情况下会相应地取整到十位、百位、千位等。示

例如下：

```
>>> a = 1627731
>>> round(a, -1)
1627730
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

3.1.3　讨论

在对值进行输出时别把取整和格式化操作混为一谈。如果只是将数值以固定的位数输出，一般来说是用不着round()的。相反，只要在格式化时指定所需要的精度就可以了。示例如下：

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:0.3f}'.format(x)
'value is 1.235'
>>>
```

此外，不要采用对浮点数取整的方式来“修正”精度上的问题。比如，我们可能会倾向于这样做：

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2)       # "Fix" result (???)



>>> c
6.3
>>>
```

对于大部分涉及浮点数的应用程序来说，一般来讲都不必（或者说不推荐）这么做。尽管这样会引入一些小误差，但这些误差是可理解的，也是可容忍的。如果说避免出现误差的行为非常重要（例如在金融类应用中），那么可以考虑使用decimal模块，这也正是下一节的主题。

3.2 执行精确的小数计算

3.2.1 问题

我们需要对小数进行精确计算，不希望因为浮点数天生的误差而带来影响。

3.2.2 解决方案

关于浮点数，一个尽人皆知的问题就是它们无法精确表达出所有的十进制小数位。此外，甚至连简单的数学计算也会引入微小的误差。例如：

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```

这些误差实际上是底层CPU的浮点运算单元和IEEE 754浮点数算术标准的一种“特性”。由于Python的浮点数类型保存的数据采用的是原始表示形式，因此如果编写的代码用到了float实例，那就无法避免这样的误差。

如果期望得到更高的精度（并愿意为此牺牲掉一些性能），可以使用decimal模块：

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>>
```

这么做初看起来似乎有点怪异（将数字以字符串的形式来指定）。但是，Decimal对象能以任何期望的方式来工作（支持所有常见的数学操作）。如果要将它们打印出来或是在字符串格式化函数中使用，它们看起来就和普通的数字一样。

decimal模块的主要功能是允许控制计算过程中的各个方面，这包括数字的位数和四舍五入。要做到这些，需要创建一个本地的上下文环境然后修改其设定。示例如下：

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.7647058823529411764705882353
>>> with localcontext() as ctx:
...     ctx.prec = 3
```

```
...     print(a / b)
...
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```

3.2.3　讨论

decimal模块实现了IBM的通用十进制算术规范（General Decimal Arithmetic Specification）。不用说，这里面有着数量庞大的配置选项，这些都超出了本书的范围。

Python新手可能会倾向于利用decimal模块来规避处理float数据类型所固有的精度问题。但是，正确理解你的应用领域是至关重要的。如果我们处理的是科学或工程类的问题，像计算机图形学或者大部分带有科学性质的问题，那么更常见的做法是直接使用普通的浮点类型。首先，在真实世界中极少有什么东西需要计算到小数点后17位（float提供17位的精度）。因此，在计算中引入的微小误差根本就不足挂齿。其次，原生的浮点数运算性能要快上许多——如果要执行大量的计算，那性能问题就显得很重要了。

也就是说我们无法完全忽略误差。数学家花费了大量的时间来研究各种算法，其中一些算法的误差处理能力优于其他的算法。我们同样还需要对类似相减抵消（subtractive cancellation）以及把大数和小数加在一起时的情况多加小心。示例如下：

```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums)     # Notice how 1 disappears




0.0
>>>
```

上面这个例子可以通过使用math.fsum()以更加精确的实现来解决：

```
>>> import math
>>> math.fsum(nums)
1.0
>>>
```

但是对于其他的算法，需要研究算法本身，并理解其误差传播（error propagation）的性质。

综上所述，decimal模块主要用在涉及像金融这一类业务的程序中。在这样的程序里，计算中如果出现微小的误差是相当令人生厌的。因此，decimal

模块提供了一种规避误差的方式。当用Python作数据库的接口时也会常常会遇到Decimal对象——尤其是当访问金融数据时更是如此。

3.3　对数值做格式化输出

3.3.1　问题

我们需要对数值做格式化输出，包括控制位数、对齐、包含千位分隔符以及其他一些细节。

3.3.2　解决方案

要对一个单独的数值做格式化输出，使用内建的format()函数即可。示例如下：

```
>>> x = 1234.56789

>>> # Two decimal places of accuracy



>>> format(x, '0.2f')
'1234.57'

>>> # Right justified in 10 chars, one-digit accuracy



>>> format(x, '>10.1f')
' 1234.6'

>>> # Left justified



>>> format(x, '<10.1f')
```

```
'1234.6 '

>>> # Centered



>>> format(x, '^10.1f')
' 1234.6 '

>>> # Inclusion of thousands separator



>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>
```

如果想采用科学计数法，只要把f改为e或者E即可，根据希望采用的指数规格来指定。示例如下：

```
>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>
```

以上两种情况中，指定宽度和精度的一般格式为'[<>^]?width[,]?(.digits)?'，这里width和digits为整数，而?代表可选的部分。同样的格式也可用于字符串的.format()方法中。示例如下：

```
>>> 'The value is {:0,.2f}'.format(x)
'The value is 1,234.57'
>>>
```

3.3.3　讨论

对数值做格式化输出通常都是很直接的。本节展示的技术既能用于浮点型数，也能适用于decimal模块中的Decimal对象。

当需要限制数值的位数时，数值会根据round()函数的规则来进行取整。示例如下：

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
'-1234.6'
>>>
```

对数值加上千位分隔符的格式化操作并不是特定于本地环境的。如果需要将这个需求纳入考虑，应该考察一下local模块中的函数。还可以利用字符串的translate()方法交换不同的分隔字符。示例如下：

```
>>> swap_separators = { ord('.'):',', ord(','):'.' }
```

```
>>> format(x, ',').translate(swap_separators)
'1.234,56789'
>>>
```

在很多Python代码中，常用%操作符来对数值做格式化处理。示例如下：

```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'    1234.6'
>>> '%-10.1f' % x
'1234.6    '
>>>
```

这种格式化操作仍然是可接受的，但是比起更加现代化的format()方法，这种方法就显得不是那么强大了。比如说，当使用%操作符来格式化数值时，有些功能就没法得到支持了（例如添加千位分隔符）。

3.4　同二进制、八进制和十六进制数打交道

3.4.1　问题

我们需要对以二进制、八进制或十六进制表示的数值做转换或输出。

3.4.2　解决方案

要将一个整数转换为二进制、八进制或十六进制的文本字符串形式，只要分别使用内建的bin()、oct()和hex()函数即可，示例如下：

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

此外，如果不希望出现0b、0o或者0x这样的前缀，可以使用format()函数。示例如下：

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

整数是有符号的，因此如果要处理负数的话，输出中也会带上一个符号。示例如下：

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
'-4d2'
>>>
```

相反，如果需要产生一个无符号的数值，需要加上最大值来设置比特位的长度。比如，要展示一个32位数，可以像这样实现：

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'11111111111111111111101100101110'
>>> format(2**32 + x, 'x')
'fffffb2e'
>>>
```

要将字符串形式的整数转换为不同的进制，只

需要使用int()函数再配合适当的进制即可。示例如下：

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

3.4.3 讨论

对于大部分的情况，处理二进制、八进制和十六进制数都是非常直接的。只是需要记住，这些转换只适用于转换整数的文本表示形式，实际在底层只有一种整数类型。

最后，对于那些用到了八进制数的程序员来说还有一个地方需要注意。在Python中指定八进制数的语法和许多其他编程语言稍有不同。比方说，如果试着做如下的操作，则会得到一个语法错误：

```
>>> import os
>>> os.chmod('script.py', 0755)
  File "<stdin>", line 1
    os.chmod('script.py', 0755)
                            ^
SyntaxError: invalid token
>>>
```

请确保在八进制数前添加0o前缀，就像这样：

```
>>> os.chmod('script.py', 0o755)
>>>
```

3.5 从字节串中打包和解包大整数

3.5.1 问题

我们有一个字节串，需要将其解包为一个整型数值。此外，还需要将一个大整数重新转换为一个字节串。

3.5.2 解决方案

假设程序需要处理一个有着16个元素的字节串，其中保存着一个128位的整数。示例如下：

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

要将字节解释为整数，可以使用int.from_bytes()，然后像这样指定字节序即可：

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

要将一个大整数重新转换为字节串，可以使用int.to_bytes()方法，只要指定字节数和字节序即可。示例如下：

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
>>>
```

3.5.3 讨论

在大整数和字节串之间互相转换并不算是常见的操作。但是，有时候在特定的应用领域中却有这样的需求，例如加密技术或网络应用中。比方说IPv6网络地址就是由一个128位的整数来表示的。如果正在编写的代码需要将这样的值从数据记录中提取出来，就得面对这个问题。

作为本节中技术的替代方案，我们可能会倾向于使用struct模块来完成解包，具体可参见6.11节。这行得通，但是struct模块可解包的整数大小是有限制的。因此，需要解包多个值，然后再将它们合并起来以得到最终的结果。示例如下：

```
>>> data
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

```
>>> import struct



>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

字节序的规范（大端或小端）指定了组成整数的字节是从低位到高位排列还是从高位到低位排列。只要我们精心构造一个十六进制数，就能很容易看出这其中的意义：

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
b'\x04\x03\x02\x01'
>>>
```

如果尝试将一个整数打包成字节串，但字节大小不合适的话就会得到一个错误信息。如果需要的话，可以使用int.bit_length()方法来确定需要用到多少位才能保存这个值：

```
>>> x = 523 ** 23
>>> x
335381300113661875107536852714019056160355655333978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if


rem:
...


nbytes += 1
...



>>>
>>> x.to_bytes(nbytes, 'little')
b'\x03X\xf1\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

3.6 复数运算

3.6.1 问题

我们的代码在同最新的Web认证方案交互时遇到了奇点（singularity）问题，而唯一的解决方案是在复平面解决。或者也许只需要利用复数完成一些计算就可以了。

3.6.2 解决方案

复数可以通过complex(real, imag)函数来指定，或者通过浮点数再加上后缀j来指定也行。示例如下：

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

实部、虚部以及共轭值可以很方便地提取出来，示例如下：

```
>>> a.real
2.0
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

此外，所有常见的算术运算操作都适用于复数：

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.4117647058823529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

最后，如果要执行有关复数的函数操作，例如求正弦、余弦或平方根，可以使用cmath模块：

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

3.6.3 讨论

Python中大部分和数学相关的模块都可适用于复数。例如，如果使用numpy模块，可以很直接地创建复数数组，并对它们执行操作：

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j, 4.+5.j, 6.-7.j, 8.+9.j])
>>> a + 2
array([ 4.+3.j, 6.+5.j, 8.-7.j, 10.+9.j])
>>> np.sin(a)
array([ 9.15449915 -4.16890696j, -56.16227422 -48.50245524j,
         -153.20827755-526.47684926j, 4008.42651446-589.49948373j
>>>
```

Python中的标准数学函数默认情况下不会产生复数值，因此像这样的值不会意外地出现在代码里。例如：

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

如果希望产生复数结果，那必须明确使用cmath模块或者在可以感知复数的库中声明对复数

类型的使用。示例如下：

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

3.7　处理无穷大和NaN

3.7.1　问题

我们需要对浮点数的无穷大、负无穷大或NaN（not a number）进行判断测试。

3.7.2　解决方案

Python中并没有特殊的语法用来表示这些特殊的浮点数值，但是它们可以通过float()来创建。示例如下：

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

要检测是否出现了这些值，可以使用math.isinf()和math.isnan()函数。示例如下：

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

3.7.3 讨论

要获得关于这些特殊的浮点数值的详细信息，应该参考IEEE 754规范。但是，这里有几个棘手的细节问题需要搞清楚，尤其是当涉及比较操作和操作符时可能出现的问题。

无穷大值在数学计算中会进行传播。例如：

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

但是，某些特定的操作会导致未定义的行为并产生NaN的结果。例如：

```
>>> a = float('inf')
>>> a/a
nan
```

```
>>> b = float('-inf')
>>> a + b
nan
>>>
```

NaN会通过所有的操作进行传播，且不会引发任何异常。例如：

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
>>> math.sqrt(c)
nan
>>>
```

有关NaN，一个微妙的特性是它们在做比较时从不会被判定为相等。例如：

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is


d
False
>>>
```

正因为如此，唯一安全检测NaN的方法是使用math.isnan()，正如本节示例代码中的那样。

有时候程序员希望在出现无穷大或NaN结果时可以修改Python的行为，让它抛出异常。fpectl模块可以用来调整这个行为，但是在标准Python中它是没有开启的，而且这个模块是同平台相关的，只针对专家级的程序员使用。可以参见Python在线文档（http://docs.python.org/3/library/fpectl.html ）以获得进一步的细节。

3.8　分数的计算

3.8.1　问题

仿佛进入时光机一样，我们突然发现自己在做涉及分数处理的小学家庭作业。或者也许我们正在为自己的木材商店编写测量计算方面的代码。

3.8.2　解决方案

fractions模块可以用来处理涉及分数的数学计算问题。示例如下：

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator



>>> c = a * b
>>> c.numerator
35
>>> c.denominator

64
>>> # Converting to a float
```

```
>>> float(c)
0.546875

>>> # Limiting the denominator of a value



>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction



>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

3.8.3 讨论

在大多数程序中，涉及分数的计算问题并不常见。但是在有些场景中使用分数还是有道理的。比如，允许程序接受以分数形式给出的单位计量并执行相应的计算，这样可以避免用户手动将数据转换为Decimal对象或浮点数。

3.9 处理大型数组的计算

3.9.1 问题

我们需要对大型的数据集比如数组或网格（grid）进行计算。

3.9.2 解决方案

对于任何涉及数组的计算密集型任务，请使用NumPy库。NumPy的主要特性是为Python提供了数组对象，比标准Python中的列表有着更好的性能表现，因此更加适合于做数学计算。下面是一个简短的示例，用来说明列表同NumPy数组在行为上的几个重要不同之处：

```
>>> # Python lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Numpy arrays
>>> import numpy as np
```

```
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6, 8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

可以看到，有关数组的几个基本数学运算在行为上都有所不同。特别是，NumPy中的数组在进行标量运算（例如 ax * 2或ax + 10）时是针对逐个元素进行计算的。此外，当两个操作数都是数组时，NumPy数组在进行数学运算时会针对数组的所有元素进行计算，并产生出一个新的数组作为结果。

由于数学操作会同时施加于所有的元素之上，这一事实使得对整个数组的计算变得非常简单和快速。比方说，如果想计算多项式的值：

```
>>> def f(x):
...     return 3*x**2 - 2*x + 7
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

NumPy提供了一些“通用函数”的集合，它们也能对数组进行操作。这些通用函数可作为math模块中所对应函数的替代。示例如下：

```
>>> np.sqrt(ax)
array([ 1.        , 1.41421356, 1.73205081, 2. ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

使用NumPy中的通用函数，其效率要比对数组进行迭代然后使用math模块中的函数每次只处理一个元素快上百倍。因此，只要有可能就应该使用这些通用函数。

在底层，NumPy数组的内存分配方式和C或者Fortran一样。即，它们是大块的连续内存，由同一种类型的数据组成。正是因为这样，NumPy才能创建比通常Python中的列表要大得多的数组。例如，如果想创建一个10000×10000的二维浮点数组，这根本不是问题：

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
       ...,
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.]])
```

```
>>>
```

所有的常用操作仍然可以同时施加于所有的元素之上：

```
>>> grid += 10
>>> grid
array([[ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.],
       ...,
       [ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.],
       [ 10., 10., 10., ..., 10., 10., 10.]])
>>> np.sin(grid)
array([[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       ...,
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111]])
>>>
```

关于NumPy，一个特别值得提起的方面就是NumPy扩展了Python列表的索引功能——尤其是针对多维数组时更是如此。为了说明，我们先构造一个简单的二维数组然后做些试验：

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> # Select row 1



>>> a[1]
array([5, 6, 7, 8])

>>> # Select column 1



>>> a[:,1]
array([ 2,  6, 10])

>>> # Select a subregion and change it



>>> a[1:3, 1:3]
array([[ 6,  7],
       [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Broadcast a row vector across an operation on all rows



>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
       [105, 117, 119, 111],
       [109, 121, 123, 115]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])
```

```
>>> # Conditional assignment on an array



>>> np.where(a < 10, a, 10)
array([[ 1, 2, 3, 4],
       [ 5, 10, 10, 8],
       [ 9, 10, 10, 10]])
>>>
```

3.9.3　讨论

Python中大量的科学和工程类函数库都以NumPy作为基础，它也是广泛使用中的最为庞大和复杂的模块之一。尽管如此，对于NumPy我们还是可以从构建简单的例子开始，逐步试验，最后实现一些有用的应用。

提到NumPy的用法，一个相对来说比较常见的导入方式是import numpy as np，正如我们给出的示例中那样，这么做缩短了名称，方便我们每次在程序中输入。

要获得更多信息，一定要去看看NumPy的官方站点http://www.numpy.org 。

3.10 矩阵和线性代数的计算

3.10.1 问题

我们需要执行矩阵和线性代数方面的操作，比如矩阵乘法、求行列式、解线性方程等。

3.10.2 解决方案

NumPy库中有一个matrix对象可用来处理这种情况。Matrix对象和3.9节中描述的数组对象有些类似，但是在计算时遵循线性代数规则。下面的例子展示了几个重要的特性：

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2, 3],
        [ 0, 4, 5],
        [ 7, 8, -9]])

>>> # Return transpose




>>> m.T
matrix([[ 1, 0, 7],
        [-2, 4, 8],
        [ 3, 5, -9]])

>>> # Return inverse
```

```
>>> m.I
matrix([[ 0.33043478, -0.02608696, 0.09565217],
        [-0.15217391, 0.13043478, 0.02173913],
        [ 0.12173913, 0.09565217, -0.0173913 ]])

>>> # Create a vector and multiply



>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
>>>
```

更多的操作可在numpy.linalg子模块中找到。例如：

```
>>> import numpy.linalg

>>> # Determinant



>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues



>>> numpy.linalg.eigvals(m)
array([-13.11474312, 2.75956154, 6.35518158])
```

```
>>> # Solve for x in mx = v



>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
matrix([[2],
        [3],
        [4]])
>>>
```

3.10.3 讨论

显然，线性代数是个庞大的课题，远超出了本书的范围。但是，如果需要处理矩阵和向量，NumPy是个很好的起点。请访问http://www.numpy.org 以获得更多详细的信息。

3.11 随机选择

3.11.1 问题

我们想从序列中随机挑选出元素，或者想生成随机数。

3.11.2 解决方案

random模块中有各种函数可用于需要随机数和随机选择的场景。例如，要从序列中随机挑选出元素，可以使用random.choice()：

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

如果想取样出N个元素，将选出的元素移除以

做进一步的考察，可以使用random.sample()：

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

如果只是想在序列中原地打乱元素的顺序（洗牌），可以使用random.shuffle()：

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

要产生随机整数，可以使用random.randint()：

```
>>> random.randint(0,10)
2
>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
```

```
10
>>> random.randint(0,10)
3
>>>
```

要产生0到1之间均匀分布的浮点数值，可以使用random.random()：

```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>
```

如果要得到由N个随机比特位所表示的整数，可以使用random.getrandbits()：

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>
```

3.11.3　讨论

random模块采用马特赛特旋转算法（Mersenne Twister，也称为梅森旋转算法）来计算随机数。这是一个确定性算法，但是可以通过random.seed()函

数来修改初始的种子值。示例如下：

```
random.seed()              # Seed based on system time or os.ura
random.seed(12345)          # Seed based on integer given
random.seed(b'bytedata')     # Seed based on byte data
```

除了以上展示的功能外，random模块还包含有计算均匀分布、高斯分布和其他概率分布的函数。比如，random.uniform()可以计算均匀分布值，而random.gauss()则可计算出正态分布值。请查阅文档以获得对其他所支持的分布的相关信息。

random模块中的函数不应该用在与加密处理相关的程序中。如果需要这样的功能，考虑使用ssl模块中的函数来替代。例如，ssl.RAND_bytes()可以用来产生加密安全的随机字节序列。

3.12 时间换算

3.12.1 问题

我们的代码需要进行简单的时间转换工作，比如将日转换为秒，将小时转换为分钟等。

3.12.2 解决方案

我们可以利用datetime模块来完成不同时间单位间的换算。例如，要表示一个时间间隔，可以像这样创建一个timedelta实例：

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

如果需要表示特定的日期和时间，可以创建

datetime实例并使用标准的数学运算来操纵它们。示例如下：

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

当执行计算时，应该要注意的是datetime模块是可正确处理闰年的。示例如下：

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

3.12.3 讨论

对于大部分基本的日期和时间操控问题，datetime模块已足够满足要求了。如果需要处理更为复杂的日期问题，比如处理时区、模糊时间范围、计算节日的日期等，可以试试dateutil模块。

为了举例说明，可以使用dateutil.relativedelta()函数完成许多同datetime模块相似的时间计算。然而，dateutil的一个显著特点是在处理有关月份的问题时能填补一些datetime模块留下的空缺（可正确处理不同月份中的天数）。示例如下：

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this funct
>>>

>>> from dateutil.relativedelta import


relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>

>>> # Time between two dates




>>> b = datetime(2012, 12, 21)
>>> d = b - a
```

```
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

3.13 计算上周5的日期

3.13.1 问题

我们希望有一个通用的解决方案能找出一周中上一次出现某天时的日期。比方说上周五是几月几号？

3.13.2 解决方案

Python的datetime模块中有一些实用函数和类可以帮助我们完成这样的计算。关于这个问题，一个优雅、通用的解决方案看起来是这样的：

```
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']

def get_previous_byday(dayname, start_date=None):
    if start_date is None:
       start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

在交互式解释器环境中使用这个函数看起来是这样的：

```
>>> datetime.today() # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

可选的start_date参数可以通过另一个datetime实例来提供。例如：

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

3.13.3 讨论

上面的解决方案将起始日期和目标日期映射到它们在一周之中的位置上（周一为第0天，依此类推）。然后用取模运算计算上一次目标日期出现时

到起始日期为止一共经过了多少天。之后，从起始日期中减去一个合适的timedelta实例就得到了我们所要的日期。

如果需要执行大量类似的日期计算，最好安装python-dateutil包。例如，下面这个例子是使用dateutil模块中的relativedelta()函数来执行同样的计算：

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday



>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday



>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

3.14 找出当月的日期范围

3.14.1 问题

我们有一些代码需要循环迭代当月中的每个日期，我们需要一种高效的方法来计算出日期的范围。

3.14.2 解决方案

对日期进行循环迭代并不需要事先构建一个包含所有日期的列表。只需计算出范围的开始和结束日期，然后在迭代时利用datetime.timedelta对象来递增日期就可以了。

下面这个函数可接受任意的datetime对象，并返回一个包含本月第一天和下个月第一天日期的元组。示例如下：

```
from datetime import

datetime, date, timedelta
import calendar
```

```
def


get_month_range(start_date=None):
    if


start_date is


None:
    start_date = date.today().replace(day=1)
_, days_in_month = calendar.monthrange(start_date.year, start_dat
end_date = start_date + timedelta(days=days_in_month)
return


(start_date, end_date)
```

当准备好这个函数后，对日期范围做循环迭代就变得非常简单了：

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while


first_day < last_day:
...     print


(first_day)
...


first_day += a_day
...
```

```
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
2012-08-08
2012-08-09
#... and so on...
```

3.14.3 讨论

上面的代码首先计算出相应月份中第一天的日期。一种快速求解的方法是利用date或者datetime对象的replace()方法，只要将属性days设为1就可以了。关于replace()方法，一个好的方面就是它创建出的对象和我们的输入对象类型是一致的。因此，如果输入是一个date实例，那得到的结果也是date实例。同样，如果输入是datetime实例，得到的也是datetime实例。

此外，我们用calendar.monthrange()函数来找出待求解的月份中有多少天。当需要得到有关日历方面的基本信息时，calendar模块都会非常有用。monthrange()是其中唯一的一个可返回元组的函数，元组中包含当月第一个工作日的日期[1]以及当月的天数（28～31）。

一旦知道了这个月中有多少天，那么结束日期就可以通过在起始日期上加上一个合适的timedelta对象来表示。尽管很微不足道，但本节给出的解决方案中一个重要的方面就是结束日期并不包含在范围内（因为它实际上是下个月的第一天）。这刚好应对了Python中切片和range操作的行为，这些操作永远不会将结束点包含在内。

要循环迭代日期范围，我们这里采用了标准的算术以及比较操作符。比如，timedelta实例可用来递增日期，而<操作符用来检查当前日期是否超过了结束日期。

最理想的方法是创建一个专门处理日期的函数，而且用法和Python内建的range()一样。幸运的是，用生成器来实现这样一个函数真的是非常容易：

```
def


date_range(start, stop, step):
    while


start < stop:
        yield


start
        start += step
```

下面是使用这个函数的示例：

```
>>> for

d in

date_range(datetime(2012, 9, 1), datetime(2012,10,1),
                            timedelta(hours=6)):
...      print

(d)
...

2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...

>>>
```

这里要再一次说明，之所以上述实现会如此简单，一个很重要的原因就在于日期和时间可以通过标准的算术和比较操作符来进行操作。

3.15 将字符串转换为日期

3.15.1 问题

我们的应用程序接收到字符串形式的临时数据，但是我们想将这些字符串转换为datetime对象，以此对它们执行一些非字符串的操作。

3.15.2 解决方案

一般来说，Python中的标准模块datetime是用来处理这种问题的简单方案。示例如下：

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

3.15.3 讨论

datetime.strptime()方法支持许多格式化代码，比如%Y代表以4位数字表示的年份，而%m代表以2

位数字表示的月份。同样值得一提的是，这些格式化占位符也可以反过来用在将datetime对象转换为字符串上。如果需要以字符串形式来表示datetime对象并且想让输出格式变得美观时，这就能派上用场了。

比如，假设有一些代码生成了datetime对象，但是需要将它们格式化为美观、方便人们阅读的日期形式，以便将其放在自动生成的信件或报告的开头处：

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

这里值得一提的是strptime()的性能通常比我们想象的还要糟糕许多，这是因为该函数是用纯Python代码实现的，而且需要处理各种各样的系统区域设定。如果要在代码中解析大量的日期，而且事先知道日期的准确格式，那么自行实现一个解决方案可能会获得巨大的性能提升。例如，如果知道日期是以“YYYY-MM-DD”的形式表示的，可以像这样自己编写一个函数：

```
from datetime import datetime
```

```
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

我们对此进行了测试，上面这个函数比datetime.strptime()快了7倍多。如果需要处理大量涉及日期的数据时，这很可能就是需要考虑的问题了。

3.16 处理涉及到时区的日期问题

3.16.1 问题

我们有一个电话会议定在芝加哥时间2012年12月21日上午9:30举行。那么在印度班加罗尔的朋友应该在当地时间几点出现才能赶上会议？

3.16.2 解决方案

对于几乎任何涉及时区的问题，都应该使用pytz模块来解决。这个Python包提供了奥尔森时区数据库，这也是许多语言和操作系统所使用的时区信息标准。

pyzt模块主要用来本地化由datetime库创建的日期。例如，下面这段代码告诉我们如何以芝加哥时间来表示日期：

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
```

```
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>
```

一旦日期经过了本地化处理，它就可以转换为其他的时区。要知道同一时间在班加罗尔是几点，可以这样做：

```
>>> # Convert to Bangalore time
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print


(bang_d)
2012-12-21 21:00:00+05:30
>>>
```

如果打算对本地化的日期做算术计算，需要特别注意夏令时转换和其他方面的细节。比如，2013年美国的标准夏令时于本地时间3月13日凌晨2点开始（此时时间要往前拨一小时）。如果直接进行算术计算就会得到错误的结果。例如：

```
>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print
```

```
(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print


(later)
2013-03-10 02:15:00-06:00      # WRONG! WRONG!
>>>
```

结果是错误的，因为上面的代码没有把本地时间中跳过的1小时给算上。要解决这个问题，可以使用timezone对象的normalize()方法。示例如下：

```
>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>
```

3.16.3　讨论

为了不让我们的头炸掉，通常用来处理本地时间的方法是将所有的日期都转换为UTC（世界统一时间）时间，然后在所有的内部存储和处理中都使用UTC时间。示例如下：

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
>>> print(utc_d)
```

```
2013-03-10 07:45:00+00:00
>>>
```

一旦转换为UTC时间，就不用担心夏令时以及其他那些麻烦事了。因此，我们可以像之前那样对日期执行普通的算术运算。如果需要将日期以本地时间输出，只需将其转换为合适的时区即可。示例如下：

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print


(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

在同时区打交道时，一个常见的问题是如何知道时区的名称？例如，在本节的示例中我们怎么知道“Asia/Kolkata”才是表示印度时间的正确时区呢？要找出时区名称，可以考察一下pyzt.country_timezones，这是一个字典，可以使用ISO 3166国家代码作为key来查询。示例如下：

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```

当读到这里的时候，根据PEP 431的描述，为了增强对时区的支持pyzt模块可能将不再建议使用。但是，本节中提到的许多建议依然是适用的（即，建议使用UTC时间等）。

---

[1] 返回值为0～6，依次代表周一到周日。——译者注

第4章　迭代器和生成器

迭代是Python中最强有力的特性之一。从高层次看，我们可以简单地把迭代看做是一种处理序列中元素的方式。但是这里还有着更多的可能，比如创建自己的可迭代对象、在itertools模块中选择实用的迭代模式、构建生成器函数等。本章的目标是解决有关迭代中的一些常见问题。

4.1 手动访问迭代器中的元素

4.1.1 问题

我们需要处理某个可迭代对象中的元素，但是基于某种原因不能也不想使用for循环。

4.1.2 解决方案

要手动访问可迭代对象中的元素，可以使用next()函数，然后自己编写代码来捕获StopIteration异常。例如，下面这个例子采用手工方式从文件中读取文本行：

```
with

open('/etc/passwd') as

f:
    try

:
        while

True:
            line = next(f)
            print
```

```
(line, end='')
    except StopIteration

:
        pass
```

一般来说，StopIteration异常是用来通知我们迭代结束的。但是，如果是手动使用next()（就像例子中那样），也可以命令它返回一个结束值，比如说None。示例如下：

```
with

open('/etc/passwd') as

f:
     while

True:
         line = next(f, None)
         if

line is

None:
             break
```

```
        print

(line, end='')
```

4.1.3　讨论

大多数情况下，我们会用for语句来访问可迭代对象中的元素。但是，偶尔也会碰到需要对底层迭代机制做更精细控制的情况。因此，了解迭代时实际发生了些什么是很有帮助的。

下面的交互式例子对迭代时发生的基本过程做了解释说明：

```
>>> items = [1, 2, 3]
>>> # Get the iterator

>>> it = iter(items)     # Invokes items.__iter__()

>>> # Run the iterator

>>> next(it)              # Invokes it.__next__()

```

```
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

本章后面的示例将对迭代技术进行扩展，因此假定读者对基本的迭代协议已有所了解。请确保将这第一个例子深深刻在脑海里。

4.2 委托迭代

4.2.1 问题

我们构建了一个自定义的容器对象，其内部持有一个列表、元组或其他的可迭代对象。我们想让自己的新容器能够完成迭代操作。

4.2.3 解决方案

一般来说，我们所要做的就是定义一个__iter__()方法，将迭代请求委托到对象内部持有的容器上。示例如下：

```
class Node:
    def __init__(self, value):
        Self._value = value
        self._children = []

def __repr__(self):
    return 'Node({!r})'.format(self._value)

def add_child(self, node):
    self._children.append(node)

def __iter__(self):
    return iter(self._children)

# Example
```

```
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    for ch in root:
        print(ch)
    # Outputs Node(1), Node(2)
```

在这个例子中，__iter__()方法只是简单地将迭代请求转发给对象内部持有的_children属性上。

4.2.4　讨论

Python的迭代协议要求__iter__()返回一个特殊的迭代器对象，由该对象实现的__next__()方法来完成实际的迭代。如果要做的只是迭代另一个容器中的内容，我们不必担心底层细节是如何工作的，所要做的就是转发迭代请求。

示例中用到的iter()函数对代码做了一定程度的简化。iter(s)通过调用s.__iter__()来简单地返回底层的迭代器，这和len(s)调用s.__len__()的方式是一样的。

4.3 用生成器创建新的迭代模式

4.3.1 问题

我们想实现一个自定义的迭代模式，使其区别于常见的内建函数（即range()、reversed ()等）。

4.3.2 解决方案

如果想实现一种新的迭代模式，可使用生成器函数来定义。这里有一个生成器可产生某个范围内的浮点数：

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

要使用这个函数，可以使用for循环对其迭代，或者通过其他可以访问可迭代对象中元素的函数（例如sum()、list()等）来使用。示例如下：

```
>>> for


n in


frange(0, 4, 0.5):
...      print


(n)
...




0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

4.3.3 讨论

函数中只要出现了yield语句就会将其转变成一个生成器。与普通函数不同，生成器只会在响应迭

代操作时才运行。这里有一个实验性的例子，我们可以试试看，以了解这样的函数的底层机制究竟是如何运转的：

```
>>> def


countdown(n):
...     print


('Starting to count from', n)
...     while


n > 0:
...             yield


n
...


n -= 1
...     print


('Done!')
...




>>> # Create the generator, notice no output appears



>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value
```

```
>>> next(c)
Starting to count from 3
3

>>> # Run to the next yield



>>> next(c)
2

>>> # Run to next yield



>>> next(c)
1

>>> # Run to next yield (iteration stops)



>>> next(c)
Done!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

这里的核心特性是生成器函数只会在响应迭代过程中的“next”操作时才会运行。一旦生成器函数返回，迭代也就停止了。但是，通常用来处理迭代的for语句替我们处理了这些细节，因此一般情况下不必为此操心。

4.4　实现迭代协议

4.4.1　问题

　　我们正在构建一个自定义的对象，希望它可以支持迭代操作，但是也希望能有一种简单的方式来实现迭代协议。

4.4.2　解决方案

　　目前来看，要在对象上实现可迭代功能，最简单的方式就是使用生成器函数。在4.2节中，我们用Node类来表示树结构。也许你想实现一个迭代器能够以深度优先的模式遍历树的节点。下面是可能的做法：

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

def __repr__(self):
    return 'Node({!r})'.format(self._value)

def add_child(self, node):
    self._children.append(node)

def __iter__(self):
    return iter(self._children)
```

```
def depth_first(self):
    yield self
    for c in self:
        yield from c.depth_first()

# Example



if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
    # Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5
```

在这份代码中，depth_first()的实现非常易于阅读，描述起来也很方便。它首先产生出自身，然后迭代每个子节点，利用子节点的depth_first()方法（通过yield from语句）产生出其他元素。

4.4.3　讨论

Python的迭代协议要求__iter__()返回一个特殊

的迭代器对象，该对象必须实现__next__()方法，并使用StopIteration异常来通知迭代的完成。但是，实现这样的对象常常会比较繁琐。例如，下面的代码展示了depth_first()的另一种实现，这里使用了一个相关联的迭代器类。

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

def __repr__(self):
    return 'Node({!r})'.format(self._value)

def add_child(self, other_node):
    self._children.append(other_node)

def __iter__(self):
    return iter(self._children)

def depth_first(self):
    return DepthFirstIterator(self)

class DepthFirstIterator(object):
'''



Depth-first traversal



'''




def __init__(self, start_node):
    self._node = start_node
    self._children_iter = None
    self._child_iter = None
```

```
def __iter__(self):
    return self

def __next__(self):
    # Return myself if just started; create an iterator for chil



    if self._children_iter is None:
        self._children_iter = iter(self._node)
        return self._node

    # If processing a child, return its next item



    elif self._child_iter:
        try:
            nextchild = next(self._child_iter)
            return nextchild
        except StopIteration:
            self._child_iter = None
            return next(self)

    # Advance to the next child and start its iteration



    else:
        self._child_iter = next(self._children_iter).depth_first
        return next(self)
```

DepthFirstIterator类的工作方式和生成器版本的实现相同但是却复杂了许多，因为迭代器必须维护迭代过程中许多复杂的状态，要记住当前迭代过程进行到哪里了。坦白说，没人喜欢编写这样令人费解的代码。把迭代器以生成器的形式来定义就皆大

欢喜了。

4.5　反向迭代

4.5.1　问题

我们想要反向迭代序列中的元素。

4.5.2　解决方案

可以使用内建的reversed()函数实现反向迭代。示例如下：

```
>>> a = [1, 2, 3, 4]
>>> for


x in


reversed(a):
...      print


(x)
...




4
3
2
1
```

反向迭代只有在待处理的对象拥有可确定的大小，或者对象实现了__reversed__()特殊方法时，才能奏效。如果这两个条件都无法满足，则必须首先将这个对象转换为列表。示例如下：

```
# Print a file backwards
f = open('somefile')
for


line in


reversed(list(f)):
    print


(line, end='')
```

请注意，像上述代码中那样将可迭代对象转换为列表可能会消耗大量的内存，尤其是当可迭代对象较大时更是如此。

4.5.3　讨论

许多程序员都没有意识到如果他们实现了__reversed__()方法，那么就可以在自定义的类上实现反向迭代。示例如下：

```
class Countdown:
```

```
def __init__(self, start):
    self.start = start
    # Forward iterator



    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Reverse iterator



    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1
```

定义一个反向迭代器可使代码变得更加高效，因为这样就无需先把数据放到列表中，然后再反向去迭代列表了。

4.6 定义带有额外状态的生成器函数

4.6.1 问题

我们想定义一个生成器函数，但是它还涉及一些额外的状态，我们希望能以某种形式将这些状态暴露给用户。

4.6.2 解决方案

如果想让生成器将状态暴露给用户，别忘了可以轻易地将其实现为一个类，然后把生成器函数的代码放到__iter__()方法中即可。示例如下：

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines,1):
        self.history.append((lineno, line))
        yield line

    def clear(self):
        self.history.clear()
```

要使用这个类，可以将其看做是一个普通的生成器函数。但是，由于它会创建一个类实例，所以可以访问内部属性，比如history属性或者clear()方法。示例如下：

```
with

open('somefile.txt') as

f:
     lines = linehistory(f)
     for

line in

lines:
         if

'python' in

line:
             for

lineno, hline in

lines.history:
                 print
```

```
('{}:{}'.format(lineno, hline), end='')
```

4.6.3　讨论

有了生成器之后很容易掉入一个陷阱，即，试着只用函数来解决所有的问题。如果生成器函数需要以不寻常的方式同程序中其他部分交互的话（比如暴露属性，允许通过方法调用来获得控制等），那就会导致出现相当复杂的代码。如果遇到了这种情况，就像示例中做的那样，用类来定义就好了。将生成器函数定义在__iter__()方法中并没有对算法做任何改变。由于状态只是类的一部分，这一事实使得我们可以很容易将其作为属性和方法来提供给用户交互。

上面所示的方法有一个潜在的微妙之处，那就是如果打算用除了for循环之外的技术来驱动迭代过程的话，可能需要额外调用一次iter()。比方说：

```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Call iter() first, then start iterating
```

```
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>
```

4.7 对迭代器做切片操作

4.7.1 问题

我们想对由迭代器产生的数据做切片处理，但是普通的切片操作符在这里不管用。

4.7.2 解决方案

要对迭代器和生成器做切片操作，itertools.islice()函数是完美的选择。示例如下：

```
>>> def


count(n):
...     while


True:
...             yield


n
...             n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in


<module>
```

```
TypeError

: 'generator' object is not

subscriptable
>>> # Now using islice()

>>> import itertools

>>> for

x in

itertools.islice(c, 10, 20):
...       print

(x)
...
10
11
12
13
14
15
16
17
18
19
>>>
```

4.7.3 讨论

迭代器和生成器是没法执行普通的切片操作的，这是因为不知道它们的长度是多少（而且它们也没有实现索引）。islice()产生的结果是一个迭代器，它可以产生出所需要的切片元素，但这是通过访问并丢弃所有起始索引之前的元素来实现的。之后的元素会由islice对象产生出来，直到到达结束索引为止。

需要重点强调的是islice()会消耗掉所提供的迭代器中的数据。由于迭代器中的元素只能访问一次，没法倒回去，因此这里就需要引起我们的注意了。如果之后还需要倒回去访问前面的数据，那也许就应该先将数据转到列表中去。

4.8 跳过可迭代对象中的前一部分元素

4.8.1 问题

我们想对某个可迭代对象做迭代处理，但是对于前面几个元素并不感兴趣，只想将它们丢弃掉。

4.8.2 解决方案

itertools模块中有一些函数可用来解决这个问题。第一个是itertools.dropwhile()函数。要使用它，只要提供一个函数和一个可迭代对象即可。该函数返回的迭代器会丢弃掉序列中的前面几个元素，只要它们在所提供的函数中返回True即可[1]。这之后，序列中剩余的全部元素都会产生出来。

为了说明，假设我们正在读取一个文件，文件的开头有一系列的注释行。示例如下：

```
>>> with


open('/etc/passwd') as
```

```
f:
...     for


line in


f:
...         print


(line, end='')
...



##
# User Database
#
# Note that this file is consulted directly only when the system
# in single-user mode. At other times, this information is provid
# Open Directory.
...



##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...



>>>
```

如果想跳过所有的初始注释行，这里有一种方法：

```
>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
```

```
...     for line in dropwhile(lambda line: line.startswith('#'),
...         print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

这个例子是根据测试函数的结果来跳过前面的元素。如果恰好知道要跳过多少个元素，那么可以使用itertools.islice()。示例如下：

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
1
4
10
15
>>>
```

在这个例子中，islice()的最后一个参数None用来表示想要前3个元素之外的所有元素，而不是只要前3个元素（即，表示切片[3:]，而不是[:3]）。

4.8.3 讨论

dropwhile()和islice()都是很方便实用的函数，

可以利用它们来避免写出如下所示的混乱代码：

```
with

open('/etc/passwd') as

f:
    # Skip over initial comments

    while

True:
        line = next(f, '')
        if not

line.startswith('#'):
            break

# Process remaining lines

while

line:
    # Replace with useful processing

    print
```

```
(line, end='')
    line = next(f, None)
```

只丢弃可迭代对象中的前一部分元素和对全部元素进行过滤也是有所区别的。例如，本节第一个示例也许可以重写为如下代码：

```
with

open('/etc/passwd') as

f:
    lines = (line for

line in

f if not

line.startswith('#'))
    for

line in

lines:
        print

(line, end='')
```

这么做显然会丢弃开始部分的注释行，但这同样会丢弃整个文件中出现的所有注释行。而本节开始给出的解决方案只会丢弃元素，直到有某个元素不满足测试函数为止。那之后的所有剩余元素全部会不经过筛选而直接返回。

最后应该要强调的是，本节所展示的技术可适用于所有的可迭代对象，包括那些事先无法确定大小的对象也是如此。这包括生成器、文件以及类似的对象。

4.9　迭代所有可能的组合或排列

4.9.1　问题

我们想对一系列元素所有可能的组合或排列进行迭代。

4.9.2　解决方案

为了解决这个问题，itertools模块中提供了3个函数。第一个是itertools.permutations()——它接受一个元素集合，将其中所有的元素重排列为所有可能的情况，并以元组序列的形式返回（即，将元素之间的顺序打乱成所有可能的情况）。示例如下：

```
>>> items = ['a', 'b', 'c']
>>> from itertools import


permutations
>>> for


p in


permutations(items):
...     print


(p)
```

```
...



('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

如果想得到较短长度的所有全排列，可以提供一个可选的长度参数。示例如下：

```
>>> for


p in


permutations(items, 2):
...     print


(p)
...



('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

使用itertools.combinations()可产生输入序列中所有元素的全部组合形式。示例如下：

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')
>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')
>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```

对于combinations()来说，元素之间的实际顺序是不予考虑的。也就是说，组合('a', 'b')和组合('b', 'a')被认为是相同的组合形式（因此只会产生出其中一种）。

当产生组合时，已经选择过的元素将从可能的候选元素中移除掉（即，如果'a'已经选过了，那么就将它从考虑范围中去掉）。itertools.combinations_with_replacement()函数解放了这一限制，允许相同的元素得到多次选择。示例如

下：

```
>>> for

c in

combinations_with_replacement(items, 3):
...     print

(c)
...

('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
('a', 'b', 'c')
('a', 'c', 'c')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>
```

4.9.3 讨论

本节只演示了一部分itertools模块的强大功能。尽管我们肯定可以自己编写代码来产生排列和组合，但这么做大概需要我们好好思考一番。当面对看起来很复杂的迭代问题时，应该总是先去查看

itertools模块。如果问题比较常见，那么很可能已经有现成的解决方案了。

4.10 以索引-值对的形式迭代序列

4.10.1 问题

我们想迭代一个序列，但是又想记录下序列中当前处理到的元素索引。

4.10.2 解决方案

内建的enumerate()函数可以非常漂亮地解决这个问题：

```
>>> my_list = ['a', 'b', 'c']
>>> for


idx, val in


enumerate(my_list):
...      print


(idx, val)
...



0 a
1 b
2 c
```

如果要打印出规范的行号（这种情况下一般是从1开始而不是0），可以传入一个start参数作为起始索引：

```
>>> my_list = ['a', 'b', 'c']
>>> for


idx, val in


enumerate(my_list, 1):
...     print


(idx, val)
...




1 a
2 b
3 c
```

这种情况特别适合于跟踪记录文件中的行号，当想在错误信息中加上行号时就特别有用了。示例如下：

```

def


parse_data(filename):
    with
```

```
open(filename, 'rt') as


f:
        for


lineno, line in


enumerate(f, 1):
            fields = line.split()
            try


:
                count = int(fields[1])
                ...
            except ValueError as


e:
                print


('Line {}: Parse error: {}'.format(lineno, e))
```

enumerate()可以方便地用来跟踪记录特定的值出现在列表中的偏移位置。比如，如果想将文件中的单词和它们所出现的行之间建立映射关系，则可以通过使用enumerate()来将每个单词映射到文件行相应的偏移位置来实现。示例如下：

```
word_summary = defaultdict(list)

with
```

```
open('myfile.txt', 'r') as


f:
    lines = f.readlines()

for


idx, line in


enumerate(lines):
    # Create a list of words in current line



    words = [w.strip().lower() for


w in


line.split()]
    for


word in


words:
        word_summary[word].append(idx)
```

处理完文件之后，如果打印word_summary，将得到一个字典（准确地说是defaultdict），而且每个单词都是字典的键。每个单词键所对应的值就是由行号组成的列表，表示这个单词曾出现过的所有行。如果单词在一行之中出现过2次，那么这个行

号就会记录2次，这使得我们可以识别出文本中各种简单的韵律。

4.10.3 讨论

对于那些可能想自己保存一个计数器的场景，enumerate()函数是个不错的替代选择，而且会更加便捷。我们可以像这样编写代码：

```
lineno = 1
for


line in


f:
    # Process line




...
lineno += 1
```

但是，通常更加优雅的做法是使用enumerate()：

```

for



lineno, line in

```

```
enumerate(f):
    # Process line


...
```

enumerate()的返回值是一个enumerate对象实例，它是一个迭代器，可返回连续的元组。元组由一个索引值和对传入的序列调用next()而得到的值组成。

尽管只是个很小的问题，这里还是值得提一下。有时候，当在元组序列上应用enumerate()时，如果元组本身也被分解展开的话就会出错。要正确处理元组序列，必须像这样编写代码：

```
data = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

# Correct!



for


n, (x, y) in


enumerate(data):
    ...
# Error!
```

```
for

n, x, y in

enumerate(data):
    ...
```

4.11 同时迭代多个序列

4.11.1 问题

我们想要迭代的元素包含在多个序列中，我们想同时对它们进行迭代。

4.11.2 解决方案

可以使用zip()函数来同时迭代多个序列。示例如下：

```
>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for


x, y in


zip(xpts, ypts):
...     print


(x,y)
...



1 101
5 78
4 37
2 15
```

```
10 62
7 99
>>>
```

zip(a, b)的工作原理是创建出一个迭代器，该迭代器可产生出元组(x, y)，这里的x取自序列a，而y取自序列b。当其中某个输入序列中没有元素可以继续迭代时，整个迭代过程结束。因此，整个迭代的长度和其中最短的输入序列长度相同。示例如下：

```
>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for


i in


zip(a,b):
...     print


(i)
...




(1, 'w')
(2, 'x')
(3, 'y')
>>>
```

如果这种行为不是所需要的，可以使用itertools.zip_longest()来替代。示例如下：

```
>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')
>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>
```

4.11.3 讨论

zip()通常用在需要将不同的数据配对在一起时。例如，假设有一列标题和一列对应的值，示例如下：

```
headers = ['name', 'shares', 'price']
values = ['ACME', 100, 490.1]
```

使用zip()，可以将这些值配对在一起来构建一个字典，就像这样：

```
s = dict(zip(headers,values))
```

此外，如果试着产生输出的话，可以编写这样的代码：

```
for

name, val in

zip(headers, values):
print

(name, '=', val)
```

尽管不常见，但是zip()可以接受多于2个序列作为输入。在这种情况下，得到的结果中元组里的元素数量和输入序列的数量相同。示例如下：

```
>>> a = [1, 2, 3]
>>> b = [10, 11, 12]
>>> c = ['x','y','z']
>>> for

i in

zip(a, b, c):
...     print
```

```
(i)
...

(1, 10, 'x')
(2, 11, 'y')
(3, 12, 'z')
>>>
```

最后需要重点强调的是，zip()创建出的结果只是一个迭代器。如果需要将配对的数据保存为列表，那么请使用list()函数。示例如下：

```
>>> zip(a, b)
<zip object at 0x1007001b8>
>>> list(zip(a, b))
[(1, 10), (2, 11), (3, 12)]
>>>
```

4.12 在不同的容器中进行迭代

4.12.1 问题

我们需要对许多对象执行相同的操作，但是这些对象包含在不同的容器内，而我们希望可以避免写出嵌套的循环处理，保持代码的可读性。

4.12.2 解决方案

itertools.chain()方法可以用来简化这个任务。它接受一系列可迭代对象作为输入并返回一个迭代器，这个迭代器能够有效地掩盖一个事实——你实际上是在对多个容器进行迭代。为了说明清楚，请考虑下面这个例子：

```
>>> from itertools import chain
>>> a = [1, 2, 3, 4]
>>> b = ['x', 'y', 'z']
>>> for x in chain(a, b):
...     print(x)
...
1
2
3
4
x
y
z
>>>
```

在程序中，chain()常见的用途是想一次性对所有的元素执行某项特定的操作，但是这些元素分散在不同的集合中。比如：

```
# Various working sets of items
active_items = set()
inactive_items = set()

# Iterate over all items
for


item in


chain(active_items, inactive_items):
    # Process item
    ...
```

采用chain()的解决方案比下面这种写两个单独的循环要优雅得多：

```

for


item in


active_items:
    # Process item
```

```
...
for

item in

inactive_items:
    # Process item

...
```

4.12.3 讨论

itertools.chain()可接受一个或多个可迭代对象作为参数，然后它会创建一个迭代器，该迭代器可连续访问并返回你提供的每个可迭代对象中的元素。尽管区别很小，但是chain()比首先将各个序列合并在一起然后再迭代要更加高效。示例如下：

```
# Inefficent
for

x in
```

```
a + b:



...

# Better
for


x in


chain(a, b):



...
```

第一种情况中，a + b操作产生了一个全新的序列，此外还要求a和b是同一种类型。chain()并不会做这样的操作，因此如果输入序列很大的话，在内存的使用上chain()就会高效得多，而且当可迭代对象之间不是同一种类型时也可以轻松适用。

4.13 创建处理数据的管道

4.13.1 问题

我们想以流水线式的形式对数据进行迭代处理（类似UNIX下的管道）。比方说我们有海量的数据需要处理，但是没法完全将数据加载到内存中去。

4.13.2 解决方案

生成器函数是一种实现管道机制的好方法。为了说明，假设我们有一个超大的目录，其中都是想要处理的日志文件：

```
foo/
   access-log-012007.gz
   access-log-022007.gz
   access-log-032007.gz
   ...
   access-log-012008
bar/
   access-log-092007.bz2
   ...
   access-log-022008
```

假设每个文件都包含如下形式的数据行：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt .
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 2
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ic
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.x
...
```

要处理这些文件，可以定义一系列小型的生成器函数，每个函数执行特定的独立任务。示例如下：

```
import os
import fnmatch
import gzip
import bz2
import re

def gen_find(filepat, top):
    '''



    Find all filenames in a directory tree that match a shell wi


    '''




    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist, filepat):
            yield os.path.join(path,name)

def gen_opener(filenames):
    '''



    Open a sequence of filenames one at a time producing a file o
```

```
    The file is closed immediately when proceeding to the next it

    '''

    for filename in filenames:
        if filename.endswith('.gz'):
            f = gzip.open(filename, 'rt')
        elif filename.endswith('.bz2'):
            f = bz2.open(filename, 'rt')
        else:
            f = open(filename, 'rt')
        yield f
        f.close()

def gen_concatenate(iterators):
    '''

    Chain a sequence of iterators together into a single sequence

    '''

    for it in iterators:
        yield from it

def gen_grep(pattern, lines):
    '''

    Look for a regex pattern in a sequence of lines
```

```
    '''



    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line):
            yield line
```

现在可以简单地将这些函数堆叠起来形成一个数据处理的管道。例如，要找出所有包含关键字python的日志行，只需要这么做：

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
for


line in


pylines:
    print


(line)
```

如果稍后想对管道进行扩展，甚至可以在生成器表达式中填充数据。比如，下面这个版本可以找出传送的字节数并统计出总字节量：

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for


line in


pylines)
bytes = (int(x) for


x in


bytecolumn if


x != '-')
print


('Total', sum(bytes))
```

4.13.3 讨论

将数据以管道的形式进行处理可以很好地适用于其他广泛的问题，包括解析、读取实时的数据源、定期轮询等。

要理解这些代码，很重要的一点是领会yield语句的含义。在这里yield语句表现为数据的生产者，而for循环表现为数据的消费者。当生成器被串联起

来时，在迭代中每个yield语句都为管道中下个阶段的处理过程产生出数据。在最后那个例子中，sum()函数实际上在驱动这整个程序，每一次都从生成器管道中取出一份数据。

这种方法的一个优点在于每个生成器函数都比较短小而且功能独立。正因为如此，编写和维护都很容易。在许多情况下，由于它们是如此的通用，因此可以在其他上下文中得到重用。最终，将这些组件粘合在一起的代码读起来就像一份食谱一样简单，因此也更容易理解。

这种方法在内存使用的高效性上也同样值得夸耀。如果目录中有着海量的文件要处理，上述展示的代码仍然可以正常工作。实际上，由于处理过程的迭代特性，这里只会用到非常少的内存。

关于gen_concatenate()函数还有一些非常微妙的地方需要说明。这个函数的目的是将输入序列连接为一个长序列行。itertools.chain()函数可以实现类似的功能，但是这需要将所有的可迭代对象指定为它的参数才行。在这个特定的例子中，这么做将涉及一行这样的代码：lines = itertools.chain(*files)，这会导致gen_opener()生成器被完全耗尽。由于这个生成器产生的是打开的文件序列，它们在下一个迭代步骤中会被立刻关闭，因

此这里不能用chain()。我们展示的解决方案避免了这个问题。

此外，gen_concatenate()函数中也出现了实现委托给一个子生成器的yield from语句。语句yield from it简单地使gen_concatenate()函数发射出所有由生成器it产生的值。这一点将在4.14节中做进一步的描述。

最后但同样重要的是，应该指出管道方法并不会总是适用于每一个数据处理问题。有时候我们需要马上处理所有的数据。但是，就算是这种情况，使用生成器管道可以在逻辑上将问题分解成一种工作流程。

David Beazley在他的“针对系统程序员之生成器技巧”教程报告（http://www.dabeaz.com/generators ）中已经对这些技术做了广泛的探讨。可以参阅他的教程以获得更多的示例。

4.14 扁平化处理嵌套型的序列

4.14.1 问题

我们有一个嵌套型的序列，想将它扁平化处理为一列单独的值。

4.14.2 解决方案

这个问题可以很容易地通过写一个带有yield from语句的递归生成器函数来解决。示例如下：

```
from collections import Iterable

def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_t
            yield from flatten(x)
        else:
            yield x

items = [1, 2, [3, 4, [5, 6], 7], 8]

# Produces 1 2 3 4 5 6 7 8



for x in flatten(items):
    print(x)
```

在上述代码中，isinstance(x, Iterable)简单地检查是否有某个元素是可迭代的。如果确实有，那么就用yield from将这个可迭代对象作为一种子例程进行递归，将它所有的值都产生出来。最后得到的结果就是一个没有嵌套的单值序列。

代码中额外的参数ignore_types和对not isinstance(x, ignore_types)的检查是为了避免将字符串和字节串解释为可迭代对象，进而将它们展开为单独的一个个字符。这使得嵌套型的字符串列表能够以大多数人所期望的方式工作。示例如下：

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for


x in


flatten(items):
...


     print


(x)
...



Dave
Paula
Thomas
Lewis
>>>
```

4.14.3 讨论

如果想编写生成器用来把其他的生成器当做子例程调用，yield from是个不错的快捷方式。如果不这么用，就需要编写有额外for循环的代码，比如这样：

```
def

flatten(items, ignore_types=(str, bytes)):
    for

x in

items:
        if

isinstance(x, Iterable) and not

isinstance(x, ignore_types):
            for

i in

flatten(x):
                yield
```

```
i
            else
:
                yield
x
```

尽管只是个小小的改变，但是使用yield from语句感觉更好，也使得代码变得更加清晰。

前面提到，对字符串和字节串的额外检查是为了避免将这些类型的对象展开为单独的字符。如果还有其他类型是不想要展开的，可以为ignore_types参数提供不同的值来确定。

最后应该要提到的是，yield from在涉及协程（coroutine）和基于生成器的并发型高级程序中有着更加重要的作用。请参见12.12节中的另一个示例。

4.15　合并多个有序序列，再对整个有序序列进行迭代

4.15.1　问题

我们有一组有序序列，想对它们合并在一起之后的有序序列进行迭代。

4.15.2　解决方案

对于这个问题，heapq.merge()函数正是我们所需要的。示例如下：

```
>>> import heapq
>>> a = [1, 4, 7, 10]
>>> b = [2, 5, 6, 11]
>>> for c in heapq.merge(a, b):
...     print(c)
...
1
2
4
5
6
7
10
11
```

4.15.3 讨论

heapq.merge的迭代性质意味着它对所有提供的序列都不会做一次性读取。这意味着可以利用它处理非常长的序列，而开销却非常小。例如，下面这个例子告诉我们如何合并两个有序的文件：

```
import heapq

with open('sorted_file_1', 'rt') as file1, \
     open('sorted_file_2') 'rt' as file2, \
     open('merged_file', 'wt') as outf:

     for line in heapq.merge(file1, file2):
         outf.write(line)
```

需要重点强调的是，heapq.merge()要求所有的输入序列都是有序的。特别是，它不会首先将所有的数据读取到堆中，或者预先做任何的排序操作。它也不会对输入做任何验证，以检查它们是否满足有序的要求。相反，它只是简单地检查每个输入序列中的第一个元素，将最小的那个发送出去。然后再从之前选择的序列中读取一个新的元素，再重复执行这个步骤，直到所有的输入序列都耗尽为止。

4.16 用迭代器取代**while**循环

4.16.1 问题

我们的代码采用while循环来迭代处理数据，因为这其中涉及调用某个函数或有某种不常见的测试条件，而这些东西没法归类为常见的迭代模式。

4.16.2 解决方案

在涉及I/O处理的程序中，编写这样的代码是很常见的：

```
CHUNKSIZE = 8192

def


reader(s):
    while


True:
        data = s.recv(CHUNKSIZE)
        if


data == b'':
            break
```

```
        process_data(data)
```

这样的代码常常可以用iter()来替换，比如：

```
def

reader(s):
    for

chunk in

iter(lambda

: s.recv(CHUNKSIZE), b''):
        process_data(data)
```

如果对这样的代码能否正常工作持有怀疑态度，可以用一个有关文件处理的类似例子试验一下：

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/
...
```

```
>>>
```

4.16.3　讨论

关于内建函数iter()，一个少有人知的特性是它可以选择性接受一个无参的可调用对象以及一个哨兵（结束）值作为输入。当以这种方式使用时，iter()会创建一个迭代器，然后重复调用用户提供的可调用对象，直到它返回哨兵值为止。

这种特定的方式对于需要重复调用函数的情况，比如这些涉及I/O的问题，有很好的效果。比如，如果想从socket或文件中按块读取数据，通常会重复调用read()或者recv()，然后紧跟着检测是否到达文件结尾。而我们给出的解决方案简单地将这两个功能合并为一个单独的iter()调用。解决方案中对lambda的使用是为了创建一个不带参数的可调用对象，但是还是可以对recv()或read()提供所需要的参数。

---

[1]　即，我们提供的那个函数起一个筛子的作用，满足条件的都会丢弃直到有元素不满足为止。——译者注

第5章　文件和I/O

任何程序都需要处理输入和输出。本章介绍了处理各种不同类型文件时的惯用方法，包括文本和二进制文件的处理、文件编码以及其他一些相关的内容。用来处理文件名和目录相关的技术也有涵盖。

5.1　读写文本数据

5.1.1　问题

我们需要对文本数据进行读写操作，但这个过程有可能针对不同的文本编码进行，比如ASCII、UTF-8或UTF-16编码。

5.1.2　解决方案

可以使用open()函数配合rt模式来读取文本文件的内容。示例如下：

```
# Read the entire file as a single string

with

open('somefile.txt', 'rt') as

f:
    data = f.read()
# Iterate over the lines of the file

with
```

```
open('somefile.txt', 'rt') as

f:
    for

line in

f:
        # process line

        ...
```

类似地，要对文本文件执行写入操作，可以使用open()函数的wt模式来完成。如果待操作的文件已存在，那么这会清除并覆盖其原先的内容。示例如下：

```
# Write chunks of text data

with

open('somefile.txt', 'wt') as
```

```
f:
    f.write(text1)



    f.write(text2)
    ...
# Redirected print statement



with


open('somefile.txt', 'wt') as


f:
    print


(line1, file=f)
    print


(line2, file=f)
    ...
```

如果要在已存在文件的结尾处追加内容，可以使用open()函数的at模式。

默认情况下，文件的读取和写入采用的都是系统默认的文本编码方式，这可以通过sys.getdefaultencoding()来查询。在大多数机器上，这项设定都被设置为utf-8。如果我们知道正在读取或写入的文本采用的是另外一种编码方式，那么可

以为open()函数提供一个可选的编码参数。示例如下：

```
with

open('somefile.txt', 'rt', encoding='latin-1') as

f:
    ...
```

Python可以识别出几百种可能的文本编码。但是，一些常见的编码方式不外乎是ascii、latin-1、utf-8以及utf-16。如果要同Web应用程序打交道，采用utf-8编码通常是比较保险的。ascii编码对应于范围U+0000到U+007F中的7比特字符。latin-1编码则是字节0～255对Unicode字符U+0000到U+00FF的直接映射。关于latin-1编码，值得注意的一点是，当读取到未知编码的文本时是不会产生解码错误的。以latin-1方式读取文件可能不会产生完全正确的解码文本，但是要从中提取出有用的数据仍然是足够了。此外，如果稍后将数据重新写入到文件中，那么原始的输入数据将得到保留。

5.1.3　讨论

一般来说，读写文本文件都是非常简单直接的。但是，这里还是有几个微妙的细节需要引起注意。首先，我们在示例中采用了with语句，这会为使用的文件创建一个上下文环境（context）。当程序的控制流程离开with语句块后，文件将自动关闭。我们并不是一定要使用with语句，但是如果不用的话请确保要记得手动关闭文件：

```
f = open('somefile.txt', 'rt')
data = f.read()
f.close()
```

另一个细微的问题是关于换行符的识别，在UNIX和Windows上它们是不同的（即，\n和\r\n之争）。默认情况下，Python工作在“通用型换行符”模式下。在该模式中，所有常见的换行格式都能识别出来。在读取时会将换行符转换成一个单独的\n字符。同样地，在输出时换行符\n会被转换为当前系统默认的换行符。如果你不想要这种“翻译”行为，可以给open()函数提供一个newline=' '的参数，示例如下：

```
# Read with disabled newline translation

with
```

```
open('somefile.txt', 'rt', newline='') as

f:
    ...
```

为了说明其中的区别，我们会在下面的例子中看到，如果在UNIX机器上读取由Windows系统编码的包含有原始数据hello world!\r\n的文本时，会出现什么结果：

```
>>> # Newline translation enabled (the default)



>>> f = open('hello.txt', 'rt')
>>> f.read()
'hello world!\n'
>>> # Newline translation disabled



>>> g = open('hello.txt', 'rt', newline='')
>>> g.read()
'hello world!\r\n'
>>>
```

最后一个问题是关于文本文件中可能出现的编码错误。当我们读取或写入文本文件时，可能会遇到编码或解码错误。例如：

```
>>> f = open('sample.txt', 'rt', encoding='ascii')
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/encodings/ascii.py", line 26, in
    return


codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in posit
12: ordinal not in range(128)
>>>
```

如果遇到这个错误，这通常表示没有以正确的编码方式来读取文件。应该仔细阅读要读取的文本的相关规范，并检查自己的操作是否正确（例如不要用latin-1编码方式读取，换成utf-8或者任何所需的编码方式）。如果还是有可能出现编码错误，则可以为open()函数提供一个可选的errors参数来处理错误。下面是几个常见的错误处理方案的例子：

```
>>> # Replace bad chars with Unicode U+fffd replacement char



>>> f = open('sample.txt', 'rt', encoding='ascii', errors='replac
>>> f.read()
'Spicy Jalape?o!'
>>> # Ignore bad chars entirely



>>> g = open('sample.txt', 'rt', encoding='ascii', errors='ignore
>>> g.read()
'Spicy Jalapeo!'
```

```
>>>
```

如果常常在摆弄open()函数的encoding和errors参数，并为此做了大量的技巧性操作（hacks），那就适得其反了，因为生活本不应该如此艰难。关于文本，第一条守则就是只需要确保总是采用正确的文本编码形式即可。当对此抱有疑问时，请使用默认的编码设定（通常是utf-8）。

5.2 将输出重定向到文件中

5.2.1 问题

我们想将print()函数的输出重定向到一个文件中。

5.2.2 解决方案

对于这个问题，只需要像这样为print()函数加上file关键字参数即可：

```
with

open('somefile.txt', 'rt') as

f:
    print

('Hello World!', file=f)
```

5.2.3 讨论

对于这个主题确实没多少东西可说。只是要确保文件是以文本模式打开的。如果文件是以二进制模式打开的话，打印就会失败。

5.3　以不同的分隔符或行结尾符完成打印

5.3.1　问题

我们想通过print()函数输出数据，但是同时也希望修改分隔符或者行结尾符。

5.3.2　解决方案

可以在print()函数中使用sep和end关键字参数来根据需要修改输出。示例如下：

```
>>> print


('ACME', 50, 91.5)
ACME 50 91.5
>>> print


('ACME', 50, 91.5, sep=',')
ACME,50,91.5
>>> print


('ACME', 50, 91.5, sep=',', end='!!\n


')
ACME,50,91.5!!
```

```
>>>
```

使用end参数也是在输出中禁止打印出换行符的方式。示例如下：

```
>>> for


i in


range(5):
...     print


(i)
...



0
1
2
3
4
>>> for


i in


range(5):
...     print


(i, end=' ')
...
```

```
0 1 2 3 4 >>>
```

5.3.3 讨论

除了空格之外，当还需要用其他字符来分隔文本时，通常在print()函数中通过sep关键字参数指定一个不同的分隔符就是最简单的方法了。有时候我们会看到有的程序员会利用str.join()来实现同样的效果。例如：

```
>>> print


(','.join('ACME','50','91.5'))
ACME,50,91.5
>>>
```

str.join()的问题就在于它只能处理字符串。这意味着我们常常得做些转换才能让其正常工作。比如说：

```
>>> row = ('ACME', 50, 91.5)
>>> print


(','.join(row))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected str instance, int found
```

```
>>> print

(','.join(str(x) for

x in

row))
ACME,50,91.5
>>>
```

其实不必这么大费周折，只要用print()函数就可以办到了：

```
>>> print

(*row, sep=',')
ACME,50,91.5
>>>
```

5.4 读写二进制数据

5.4.1 问题

我们需要读写二进制数据，比如图像、声音文件等。

5.4.2 解决方案

使用open()函数的rb或者wb模式就可以实现对二进制数据的读或写。示例如下：

```
# Read the entire file as a single byte string

with

open('somefile.bin', 'rb') as

f:
    data = f.read()
# Write binary data to a file

with

open('somefile.bin', 'wb') as
```

```
f:
    f.write(b'Hello World')
```

当读取二进制数据时，很重要的一点是所有的数据将以字节串（byte string）的形式返回，而不是文本字符串。同样地，当写入二进制数据时，数据必须是以对象的形式来提供，而且该对象可以将数据以字节形式暴露出来（即，字节串、bytearray对象等）。

5.4.3 讨论

当读取二进制数据时，由于字节串和文本字符串之间存在微妙的语义差异，这可能会造成一些潜在的问题。特别要注意的是，在做索引和迭代操作时，字节串会返回代表该字节的整数值而不是字符串。示例如下：

```
>>> # Text string
>>> t = 'Hello World'
>>> t[0]
'H'
```

```
>>> for

c in

t:
...     print

(c)
...


H
e
l
l
o
...


>>> # Byte string


>>> b = b'Hello World'
>>> b[0]
72
>>> for

c in

b:
...     print

(c)
...
```

```
72
101
108
108
111
...

>>>
```

如果需要在二进制文件中读取或写入文本内容，请确保要进行编码或解码操作。示例如下：

```
with open('somefile.bin', 'rb') as f:
    data = f.read(16)
    text = data.decode('utf-8')

with open('somefile.bin', 'wb') as f:
    text = 'Hello World'
    f.write(text.encode('utf-8'))
```

关于二进制I/O，一个鲜为人知的行为是，像数组和C结构体这样的对象可以直接用来进行写操作，而不必先将其转换为byte对象。示例如下：

```
import array

nums = array.array('i', [1, 2, 3, 4])
with

open('data.bin','wb') as

f:
    f.write(nums)
```

这种行为可适用于任何实现了所谓的“缓冲区接口（buffer interface）”的对象。该接口直接将对象底层的内存缓冲区暴露给可以在其上进行的操作。写入二进制数据就是这样一种操作。

有许多对象还支持直接将二进制数据读入到它们底层的内存中，只要使用文件对象的readinto()方法就可以了。示例如下：

```
>>> import array
```

```
>>> a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])
>>> with


open('data.bin', 'rb') as


f:
...


f.readinto(a)
...




16
>>> a
array('i', [1, 2, 3, 4, 0, 0, 0, 0])
>>>
```

但是，使用这项技术时需要特别小心，因为这常常是与平台特性相关的，而且可能依赖于字（word）的大小和字节序（即大端和小端）等属性。请参见5.9节中的另一个例子，在该例中我们将二进制数据读入到一个可变缓冲区（mutable buffer）中。

5.5　对已不存在的文件执行写入操作

5.5.1　问题

我们想将数据写入到一个文件中，但只在该文件已不在文件系统中时才这么做。

5.5.2　解决方案

这个问题可以通过使用open()函数中鲜为人知的x模式替代常见的w模式来解决。示例如下：

```
>>> with


open('somefile', 'wt') as


f:
...


f.write('Hello\n


')
...
```

```
>>> with


open('somefile', 'xt') as


f:
...


f.write('Hello\n


')
...



Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'somefile'
>>>
```

如果文件是二进制模式的，那么用xb模式代替xt即可。

5.5.3 讨论

本节中的示例以一种非常优雅的方式解决了一个常会在写文件时出现的问题（即，意外地覆盖了某个已存在的文件）。另一种解决方案是首先像这样检查文件是否已存在：

```
>>> import os
```

```
>>> if not

os.path.exists('somefile'):
...     with

open('somefile', 'wt') as

f:
...

f.write('Hello\n

')
...     else

:
...         print

('File already exists!')
...



File already exists!
>>>
```

很明显，使用x模式更加简单直接。需要注意的是，x模式是Python 3中对open()函数的扩展。在早期的Python版本或者在Python的实现中用到的底

层C函数库里都不存在这样的模式。

5.6　在字符串上执行I/O操作

5.6.1　问题

我们想将一段文本或二进制字符串写入类似于文件的对象上。

5.6.2　解决方案

使用io.StringIO()和io.BytesIO()类来创建类似于文件的对象，这些对象可操作字符串数据。示例如下：

```
>>> s = io.StringIO()
>>> s.write('Hello World\n


')
12
>>> print


('This is a test', file=s)
15
>>> # Get all of the data written so far



>>> s.getvalue()
'Hello World\nThis is a test\n'
>>>
>>> # Wrap a file interface around an existing string
```

```
>>> s = io.StringIO('Hello\n

World\n

')
>>> s.read(4)
'Hell'
>>> s.read()
'o\nWorld\n'
>>>
```

io.StringIO类只能用于对文本的处理。如果要操作二进制数据，请使用io.BytesIO。示例如下：

```
>>> s = io.BytesIO()
>>> s.write(b'binary data')
>>> s.getvalue()
b'binary data'
>>>
```

5.6.3　讨论

当出于某种原因需要模拟出一个普通文件时，这种情况下StringIO和BytesIO类是最为适用的。例如，在单元测试中，可能会使用StringIO来创建一个文件型的对象，对象中包含了测试用的数据。之

后我们可将这个对象发送给一个可以接受普通文件的函数。

请注意，StringIO和BytesIO实例是没有真正的文件描述符来对应的。因此，它们没法工作在需要一个真正的系统级文件例如文件、管道或套接字的代码环境中。

5.7 读写压缩的数据文件

5.7.1 问题

我们需要读写以gzip或bz2格式压缩过的文件中的数据。

5.7.2 解决方案

gzip和bz2模块使得同这类压缩型文件打交道变得非常简单。这两个模块都提供了open()的其他实现，可用于处理压缩文件。例如，要将压缩文件以文本形式读取，可以这样处理：

```
# gzip compression

import gzip

with

gzip.open('somefile.gz', 'rt') as

f:
    text = f.read()
# bz2 compression
```

```
import bz2

with

bz2.open('somefile.bz2', 'rt') as

f:
    text = f.read()
```

与之相似，要写入压缩数据，可以这样处理：

```
# gzip compression
import gzip

with

gzip.open('somefile.gz', 'wt') as f:
    f.write(text)

# bz2 compression
import bz2

with

 bz2.open('somefile.bz2', 'wt') as f:
    f.write(text)
```

如示例代码所示，以上所有的I/O操作都会采用文本形式并执行Unicode编码/解码操作。如果想处理二进制数据，请使用rb或wb模式。

5.7.3 讨论

大部分情况下读写压缩数据都是简单而直接的。但是请注意，选择正确的文件模式是至关重要的。如果没有指定模式，那么默认的模式是二进制，这会使得期望接受文本的程序崩溃。gzip.open()和bz2.open()所接受的参数与内建的open()函数一样，也支持encoding、errors、newline等关键字参数。

当写入压缩数据时，压缩级别可以通过compresslevel关键字参数来指定，这是可选的。示例如下：

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
    f.write(text)
```

默认级别是9，代表着最高的压缩等级。低等级的压缩可带来更好的性能表现，但压缩比就没有那么大。

最后，gzip.open()和bz2.open()有一个较少提到的特性，那就是它们能够对已经以二进制模式打开的文件进行叠加操作。示例如下：

```
import gzip



f = open('somefile.gz', 'rb')
with


gzip.open(f, 'rt') as


g:
    text = g.read()
```

这种行为使得gzip和bz2模块可以同各种类型的类文件对象比如套接字、管道和内存文件一起工作。

5.8　对固定大小的记录进行迭代

5.8.1　问题

与其按行来迭代文件，我们想对一系列固定大小的记录或数据块进行迭代。

5.8.2　解决方案

可以利用iter()和functools.partial()来完成这个巧妙的技巧，示例如下：

```
from functools import


partial
RECORD_SIZE = 32
with


open('somefile.data', 'rb') as


f:
    records = iter(partial(f.read, RECORD_SIZE), b'')
    for


r in


records:
```

```
...
```

示例中的records对象是可迭代的，它会产生出固定大小的数据块直到到达文件结尾。但是请注意，如果文件大小不是记录大小的整数倍的话，那么最后产生出的那个数据块可能比所期望的字节数要少。

5.8.3　讨论

关于iter()函数，一个少有人知的特性是，如果传递一个可调用对象及一个哨兵值给它，那么它可以创建出一个迭代器。得到的迭代器会重复调用用户提供的可迭代对象，直到返回的值为哨兵值为止，此时迭代过程停止。

在我们给出的解决方案中，functools.partial用来创建可调用对象，每次调用它时都从文件中读取固定的字节数。b' '在这里用作哨兵值，当读取到文件结尾时就会返回这个值，此时迭代过程结束。

最后但也很重要的是，解决方案中展示的文件是以二进制模式打开的。对于读取固定大小的记

录，这恐怕是最为常见的情况了。如果要针对文本文件，那么按行读取（默认的迭代行为）更为普遍一些。

5.9　将二进制数据读取到可变缓冲区中

5.9.1　问题

我们想将二进制数据直接读取到一个可变缓冲区中，中间不经过任何拷贝环节。也许我们想原地修改数据再将它写回到文件中去。

5.9.2　解决方案

要将数据读取到可变数组中，使用文件对象的readinto()方法即可。示例如下：

```
import os.path

def

read_into_buffer(filename):
    buf = bytearray(os.path.getsize(filename))
    with

open(filename, 'rb') as

f:
```

```
        f.readinto(buf)
    return

buf
```

下面来演示这个函数的用法：

```
>>> # Write a sample file


>>> with

open('sample.bin', 'wb') as

f:
...

f.write(b'Hello World')
...


>>> buf = read_into_buffer('sample.bin')
>>> buf
bytearray(b'Hello World')
>>> buf[0:5] = b'Hallo'
>>> buf
bytearray(b'Hallo World')
>>> with

open('newsample.bin', 'wb') as

f:
```

```
...

f.write(buf)
...

11
>>>
```

5.9.3 讨论

文件对象的readinto()方法可用来将数据填充到任何预分配好的数组中，这包括array模块或者numpy这样的库所创建的数组。与普通的read()方法不同的是，readinto()是为已存在的缓冲区填充内容，而不是分配新的对象然后再将它们返回。因此，可以用readinto()来避免产生额外的内存分配动作。例如，如果正在读取一个由相同大小的记录所组成的二进制文件，可以像这样编写代码：

```
record_size = 32         # Size of each record (adjust value)

buf = bytearray(record_size)
with

open('somefile', 'rb') as

f:
```

```
while


True:
    n = f.readinto(buf)
    if


n < record_size:
        break



    # Use the contents of buf



    ...
```

这里用到的另一个有趣的特性应该就是内存映像（memoryview）了，它使得我们可以对已存在的缓冲区做切片处理，但是中间不涉及任何拷贝操作，我们甚至还可以修改它的内容。示例如下：

```
>>> buf
bytearray(b'Hello World')
>>> m1 = memoryview(buf)
>>> m2 = m1[-5:]
>>> m2
<memory at 0x100681390>
>>> m2[:] = b'WORLD'
>>> buf
bytearray(b'Hello WORLD')
>>>
```

使用f.readinto()需要注意的一点是，必须总是确保要检查它的返回值，即实际读取的字节数。

如果字节数小于所提供的缓冲区大小，这可能表示数据被截断或遭到了破坏（例如，如果期望读取到一个准确的字节数时）。

最后，可以在各种库模块找到那些带有“into”的函数（例如recv_into()、pack_into()等）。Python中有许多模块都已经支持直接I/O访问了，可用来填充或修改数组和缓冲区中的内容。

请参见6.12节中那个解释二进制结构体和memoryview用法的示例，那个例子明显要更加高级一些。

5.10 对二进制文件做内存映射

5.10.1 问题

我们想通过内存映射的方式将一个二进制文件加载到可变的字节数组中，这样可以随机访问其内容，或者是实现就地修改。

5.10.2 解决方案

可以使用mmap模块实现对文件的内存映射操作。下面给出一个实用函数，以可移植的方式演示如何打开一个文件并对它进行内存映射操作：

```
import os

import mmap

def

memory_map(filename, access=mmap.ACCESS_WRITE):
    size = os.path.getsize(filename)
    fd = os.open(filename, os.O_RDWR)
    return
```

```
mmap.mmap(fd, size, access=access)
```

要使用这个函数，需要准备一个已经创建好的文件并为之填充一些数据。下面的例子告诉我们如何创建一个初始文件，然后将其扩展为所需要的大小：

```
>>> size = 1000000
>>> with


open('data', 'wb') as


f:
...


f.seek(size-1)
...


f.write(b'\x00


')
...



>>>
```

下面是用memory_map()函数对文件内容做内存

映射操作的例子：

```
>>> m = memory_map('data')
>>> len(m)
1000000
>>> m[0:10]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> m[0]
0
>>> # Reassign a slice



>>> m[0:11] = b'Hello World'
>>> m.close()
>>> # Verify that changes were made



>>> with


open('data', 'rb') as


f:
...     print


(f.read(11))
...



b'Hello World'
>>>
```

由mmap()返回的mmap对象也可以当做上下文

管理器使用，在这种情况下，底层的文件会自动关闭。示例如下：

```
>>> with


memory_map('data') as


m:
...     print


(len(m))
...     print


(m[0:10])
...




1000000
b'Hello World'
>>> m.closed
True
>>>
```

默认情况下，memory_map()函数打开的文件既可以读也可以写。对数据的任何修改都会拷贝回原始的文件中。如果需要只读访问，可以为access参数提供mmap.ACCESS_READ值。示例如下：

```
m = memory_map(filename, mmap.ACCESS_READ)
```

如果只想在本地修改数据，并不想将这些修改写回到原始文件中，可以使用mmap.ACCESS_COPY参数：

```
m = memory_map(filename, mmap.ACCESS_COPY)
```

5.10.3 讨论

通过mmap将文件映射到内存中后，我们能够以高效和优雅的方式对文件的内容进行随机访问。比方说，与其打开文件后通过组合各种seek()、read()和write()调用来访问，不如简单地将文件映射到内存，然后通过切片操作来访问数据。

通常，由mmap()暴露出的内存看起来就像一个bytearray对象。但是，利用memoryview能够以不同的方式来解读数据。比如：

```
>>> m = memory_map('data')
>>> # Memoryview of unsigned integers
>>> v = memoryview(m).cast('I')
```

```
>>> v[0] = 7
>>> m[0:4]
b'\x07\x00\x00\x00'
>>> m[0:4] = b'\x07\x01\x00\x00'
>>> v[0]
263
>>>
```

应该强调的是，对某个文件进行内存映射并不会导致将整个文件读到内存中。也就是说，文件并不会拷贝到某种内存缓冲区或数组上。相反，操作系统只是为文件内容保留一段虚拟内存而已。当访问文件的不同区域时，文件的这些区域将被读取并按照需要映射到内存区域中。但是，文件中从未访问过的部分会简单地留在磁盘上。这一切都是以透明的方式在幕后完成的。

如果有多个Python解释器对同一个文件做了内存映射，得到的mmap对象可用来在解释器之间交换数据。也就是说，所有的解释器可以同时读/写数据，在一个解释器中对数据做出的修改会自动反映到其他的解释器上。很明显，这里需要一些额外的步骤来处理同步问题，但是有时候可用这种方法作为通过管道或socket传输数据的替代方式。

本节中的示例已经尽量以通用的形式实现，能够在UNIX和Windows上都适用。请注意，对于mmap()的使用，不同的平台上会存在一些差异。此外，还有选项可用来创建匿名的内存映射区域。如果对此感兴趣，请确保仔细阅读有关这个主题的Python文档（http://docs.python.org/3/library/mmap.html ）。

5.11 处理路径名

5.11.1 问题

我们需要处理路径名以找出基文件名、目录名、绝对路径等相关的信息。

5.11.2 解决方案

要操纵路径名，可以使用os.path模块中的函数。下面是一个交互式的例子，用来说明其中一些核心的功能：

```
>>> import os



>>> path = '/Users/beazley/Data/data.csv'

>>> # Get the last component of the path



>>> os.path.basename(path)
'data.csv'
>>> # Get the directory name



>>> os.path.dirname(path)
'/Users/beazley/Data'
```

```
>>> # Join path components together



>>> os.path.join('tmp', 'data', os.path.basename(path))
'tmp/data/data.csv'

>>> # Expand the user's home directory



>>> path = '~/Data/data.csv'
>>> os.path.expanduser(path)
'/Users/beazley/Data/data.csv'

>>> # Split the file extension



>>> os.path.splitext(path)
('~/Data/data', '.csv')
>>>
```

5.11.3 讨论

对于任何需要处理文件名的问题，都应该使用os.path模块而不是通过使用标准的字符串操作来自己实现这部分功能。部分原因是为了考虑可移植性。os.path模块知道UNIX和Windows系统之间的一些差异，能够可靠地处理类似Data/data.csv和Data\data.csv这样的文件名。其次，我们真的不应该花时间去重造轮子。通常最好是直接使用那些已经提供了的功能。

应该值得一提的是，os.path模块中还有许多功能没有在本节中展示出来。可以参阅文档以获得更多同文件测试、符号链接等功能相关的函数。

5.12　检测文件是否存在

5.12.1　问题

我们需要检测某个文件或目录是否存在。

5.12.2　解决方案

可以通过os.path模块来检测某个文件或目录是否存在。示例如下：

```
>>> import os



>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('/tmp/spam')
False
>>>
```

之后可以执行进一步的测试来查明这个文件的类型。如果文件不存在的话，下面这些检测就会返回False：

```
>>> # Is a regular file
```

```
>>> os.path.isfile('/etc/passwd')
True

>>> # Is a directory



>>> os.path.isdir('/etc/passwd')
False

>>> # Is a symbolic link



>>> os.path.islink('/usr/local/bin/python3')
True

>>> # Get the file linked to



>>> os.path.realpath('/usr/local/bin/python3')
'/usr/local/bin/python3.3'
>>>
```

如果需要得到元数据（即，文件大小或修改日期），这些功能在os.path模块中也有提供：

```
>>> os.path.getsize('/etc/passwd')
3669
>>> os.path.getmtime('/etc/passwd')
1272478234.0
>>> import time



>>> time.ctime(os.path.getmtime('/etc/passwd'))
```

```
'Wed Apr 28 13:10:34 2010'
>>>
```

5.12.3 讨论

利用os.path模块来对文件做检测是简单而直接的。也许在编写脚本时唯一需要注意的事情就是关于权限的问题了——尤其是获取元数据的操作。比如：

```
>>> os.path.getsize('/Users/guido/Desktop/foo.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/genericpath.py", line 49, in get
    return


os.stat(filename).st_size
PermissionError: [Errno 13] Permission denied: '/Users/guido/Desk
>>>
```

5.13　获取目录内容的列表

5.13.1　问题

我们想获取文件系统中某个目录下所包含的文件列表。

5.13.2　解决方案

可以使用os.listdir()函数来获取目录中的文件列表。示例如下：

```
import os

names = os.listdir('somedir')
```

这么做会得到原始的目录文件列表，包括所有的文件、子目录、符号链接等。如果需要以某种方式来筛选数据，可以考虑利用列表推导式结合os.path模块中的各种函数来完成。示例如下：

```
import os.path

# Get all regular files

names = [name for

name in

os.listdir('somedir')
         if

os.path.isfile(os.path.join('somedir', name))]
# Get all dirs

dirnames = [name for

name in

os.listdir('somedir')
            if

os.path.isdir(os.path.join('somedir', name))]
```

字符串的startswith()和endswith()方法对于筛选目录中的内容也同样有用。比如:

```
pyfiles = [name for
```

```
name in

os.listdir('somedir')
           if

name.endswith('.py')]
```

至于文件名的匹配，可能会想到用glob或者fnmatch模块。示例如下：

```
import glob


pyfiles = glob.glob('somedir/*.py')

from fnmatch import

fnmatch
pyfiles = [name for

name in

os.listdir('somedir')
           if

fnmatch(name, '*.py')]
```

5.13.3 讨论

得到目录中内容的列表很简单，但是这只会带来目录中每个条目的名称。如果想得到一些附加的元数据，比如文件大小、修订日期等，要么使用os.path模块中的其他函数，要么使用os.stat()函数。要收集这些数据，请参见示例：

```
# Example of getting a directory listing

import os

import os.path

import glob

pyfiles = glob.glob('*.py')
# Get file sizes and modification dates

name_sz_date = [(name, os.path.getsize(name), os.path.getmtime(na
                for
```

```
name in

pyfiles]
for

name, size, mtime in

name_sz_date:
    print

(name, size, mtime)
# Alternative: Get file metadata

file_metadata = [(name, os.stat(name)) for

name in

pyfiles]
for

name, meta in

file_metadata:
    print

(name, meta.st_size, meta.st_mtime)
```

最后但也很重要的是，请注意有关文件名编码时会出现的一些微妙问题。一般来说，像os.listdir()这样的函数返回的条目都会根据系统默认的文件名编码方式来进行解码处理。但是，有可能在特定的条件下会遇到无法解码的文件名。5.14节和5.15节中有更多关于处理这样的名称时应该注意的细节。

5.14 绕过文件名编码

5.14.1 问题

我们想对使用了原始文件名的文件执行I/O操作，这些文件名没有根据默认的文件名编码规则来解码或编码。

5.14.2 解决方案

默认情况下，所有的文件名都会根据sys.getfilesystemencoding()返回的文本编码形式进行编码和解码。例如：

```
>>> sys.getfilesystemencoding()
'utf-8'
>>>
```

如果基于某些原因想忽略这种编码，可以使用原始字节串来指定文件名。示例如下：

```
>>> # Wrte a file using a unicode filename



>>> with
```

```
open('jalape\xf1

o.txt', 'w') as

f:
...

f.write('Spicy!')
...



6
>>> # Directory listing (decoded)



>>> import os



>>> os.listdir('.')
['jalapeño.txt']

>>> # Directory listing (raw)



>>> os.listdir(b'.')      # Note: byte string



[b'jalapen\xcc\x83o.txt']

>>> # Open file with raw filename



>>> with
```

```
open(b'jalapen\xcc\x83

o.txt') as

f:
...     print

(f.read())
...

Spicy!
>>>
```

在上两个操作中可以看到，当给同文件相关的函数比如open()和os.listdir()提供字节串参数时，对文件名的处理就发生了微小的改变。

5.14.3 讨论

一般情况下，不应该去担心有关文件名编码和解码的问题——普通的文件名操作应该能正常工作。但是，有许多操作系统可能会允许用户通过意外或恶意的方式创建出文件名不遵守期望的编码规则的文件。这样的文件名可能会使得处理大量文件的Python程序莫名其妙地崩溃。

在读取目录和同文件名打交道时，以原始的未解码的字节作为文件名就可以避免这样的问题，只是编程的时候要麻烦一些。

请参见5.15节中关于打印出无法解码的文件名的相关示例。

5.15 打印无法解码的文件名

5.15.1 问题

我们的程序接收到一个目录内容的列表，但是当程序试着打印出文件名时，会出现UnicodeEncodeError异常并伴随着一条难以理解的提示信息："不允许代理（surrogates not allowed）"，然后程序就崩溃了。

5.15.2 解决方案

当打印来路不明的文件名时，可以使用下面的方式来避免出现错误：

```
def


bad_filename(filename):
    return


repr(filename)[1:-1]
try


:
    print
```

```
(filename)
except UnicodeEncodeError

:
    print

(bad_filename(filename))
```

5.15.3　讨论

当程序必须去操纵文件系统时，本节提到了一个一般情况下很罕见但却非常令人头疼的问题。默认情况下，Python假设所有的文件名都是根据sys.getfilesystemencoding()返回的编码形式进行编码的。但是，某些文件系统不一定会强制执行这种编码约束，因此会允许文件以不恰当的编码方式来命名。这并不常见，但是总有某些用户会做出些愚蠢的事情，意外地创建出这么一个文件来（即，可能在某些有问题的代码中将不恰当的文件名传给open()）。因此危险总是存在的。

当执行类似os.listdir()这样的命令时，错误的文件名会使Python陷入窘迫的境地。一方面Python不能直接丢弃错误的名字，而另一方面它也无法将文件名转为合适的文本字符串。对于这个问题，Python的解决方案是在文件名中取出一个无法解码

的字节值\xhh，将其映射到一个所谓的“代理编码（surrogate encoding）”中，代理编码由Unicode字符\udchh来表示。参见下面的示例，在一个有缺陷的目录列表中包含着一个名为bäd.txt的文件，该文件名的编码方式为Latin-1而不是UTF-8，我们来看看显示出来的结果：

```
>>> import os



>>> files = os.listdir('.')
>>> files
['spam.py', 'b\udce4d.txt', 'foo.txt']
>>>
```

如果代码只是用来操纵文件名或者甚至是将文件名传递给函数（比如open()），一切都能正常工作。只有当想把文件名输出时才会陷入麻烦（即，打印到屏幕、记录到日志上等）。具体而言，如果试着打印上面这个列表，程序就会崩溃：

```
>>> for


name in


files:
...     print
```

```
(name)
...



spam.py
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udce4
position 1: surrogates not allowed
>>>
```

崩溃的原因在于字符\udce4不是合法的Unicode字符。它实际上是2字符组合的后半部分，这个组合称为代理对（surrogate pair）。但是，由于前半部分丢失了，因此是非法的Unicode。所以，唯一能成功产生输出的方式是，当遇到有问题的文件名时采取纠正措施。比如，将代码改为下面的方式就能产生出结果了：

```
>>> for

name in

files:
...     try

:
...
```

```
            print

(name)
...     except UnicodeEncodeError

:
...             print

(bad_filename(name))
...



spam.py
b\udce4d.txt
foo.txt
>>>
```

函数bad_filename()要实现什么功能很大程度上取决于自己的选择。比如，另一种选择是以其他方式重新编码，就像这样：

```
def

bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), errors='s
    return

temp.decode('latin-1')
```

如果使用上面这个版本的bad_filename()，就会产生如下的输出：

```
>>> for

name in

files:
...

    try

:
...

            print

(name)
...

    except UnicodeEncodeError

:
...

      print

(bad_filename(name))
...
spam.py
bäd.txt
foo.txt
>>>
```

大部分读者可能都会忽略这一节的内容。但是，如果要编写完成关键任务的脚本，需要可靠地与文件名以及文件系统打交道，那么就需要好好考虑本节的内容。否则，可能就需要周末被叫去办公室调试一个看似无法理解的错误。

5.16　为已经打开的文件添加或修改编码方式

5.16.1　问题

我们想为一个已经打开的文件添加或修改Unicode编码，但不必首先将其关闭。

5.16.2　解决方案

如果想为一个以二进制模式打开的文件对象添加Unicode编码/解码，可以用io.TextIOWrapper()对象将其包装。示例如下：

```
import urllib.request

import io

u = urllib.request.urlopen('http://www.python.org')
f = io.TextIOWrapper(u,encoding='utf-8')
text = f.read()
```

如果想修改一个已经以文本模式打开的文件的编码方式，可以在用新的编码替换之前的编码前，用detach()方法将已有的文本编码层移除。下面是修改sys.stdout编码的例子：

```
>>> import sys



>>> sys.stdout.encoding
'UTF-8'
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding=
>>> sys.stdout.encoding
'latin-1'
>>>
```

这么做可能会破坏终端上的输出，这里只是用做说明使用。

5.16.3　讨论

I/O系统是以一系列的层次来构建的。我们可以通过下面这个涉及文本文件的简单例子来观察这些层次：

```
>>> f = open('sample.txt','w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedWriter name='sample.txt'>
>>> f.buffer.raw
```

```
<_io.FileIO name='sample.txt' mode='wb'>
>>>
```

在这个例子中，io.TextIOWrapper是一个文本处理层，它负责编码和解码Unicode。而io.BufferedWriter是一个缓冲I/O层，负责处理二进制数据。最后，io.FileIO是一个原始文件，代表着操作系统底层的文件描述符。添加或修改文本的编码涉及添加或修改最上层的io.TextIOWrapper层。

作为一般的规则，直接通过访问上面展示的属性来操纵不同的层次是不安全的。比如，如果用这种技术来修改编码的话，看看会出现什么情况：

```
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>> f.write('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in


<module>
ValueError


: I/O operation on closed file.
>>>
```

这根本不起作用，因为f之前的值已经被销毁，在这个过程中导致底层的文件被关闭。

detach()方法将最上层的io.TextIOWrapper层同文件分离开来，并将下一个层次（io.BufferedWriter）返回。在这之后，最上层将不再起作用。示例如下：

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> b = f.detach()
>>> b
<_io.BufferedWriter name='sample.txt'>
>>> f.write('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>>
```

一旦完成分离，就可以为返回的结果添加一个新的最上层。示例如下：

```
>>> f = io.TextIOWrapper(b, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>>
```

尽管这里我们已经展示了如何修改编码方式，其实也可以利用这项技术来修改文本行的处理、错

误处理机制以及其他有关文件处理方面的行为。示例如下：

```
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding=
...


errors='xmlcharrefreplace')
>>> print


('Jalape\u00f1


o')
Jalapeño
>>>
```

在输出中，我们注意到非ASCII字符ñ已经被&#241所取代了。

5.17 将字节数据写入文本文件

5.17.1 问题

我们想将一些原始字节写入到以文本模式打开的文件中。

5.17.2 解决方案

只需要简单的将字节数据写入到文件底层的buffer中就可以了。示例如下：

```
>>> import sys



>>> sys.stdout.write(b'Hello\n


')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>> sys.stdout.buffer.write(b'Hello\n


')
Hello
5
>>>
```

同样地，我们也可以从文本文件中读取二进制数据，只要通过buffer属性来读取即可。

5.17.3 讨论

I/O系统是以不同的层次来构建的。文本文件是通过在缓冲的二进制模式文件之上添加一个Unicode编码/解码层来构建的。buffer属性简单地指向底层的文件。如果访问该属性，就可以绕过文本编码/解码层了。

例子中的sys.stdout可以被视为特殊情况。默认情况下，sys.stdout总是以文本模式打开的。但是，如果要编写一个需要将二进制数据转储到标准输出的脚本，就可以使用上面演示的技术来绕过文本编码层。

5.18　将已有的文件描述符包装为文件对象

5.18.1　问题

我们有一个以整数值表示的文件描述符，它已经同操作系统中已打开的I/O通道建立起了联系（即，文件、管道、socket等）。而我们希望以高级的Python文件对象来包装这个文件描述符。

5.18.2　解决方案

文件描述符与一般打开的文件相比是有区别的。区别在于，文件描述符只是一个由操作系统分配的整数句柄，用来指代某种系统I/O通道。如果刚好有这样一个文件描述符，就可以通过open()函数用Python文件对象对其进行包装。这很简单，只需将整数形式的文件描述符作为第一个参数取代文件名就可以了。示例如下：

```
# Open a low-level file descriptor

import os
```

```
fd = os.open('somefile.txt', os.O_WRONLY | os.O_CREAT)
# Turn into a proper file



f = open(fd, 'wt')
f.write('hello world\n


')
f.close()
```

当高层的文件对象被关闭或销毁时，底层的文件描述符也会被关闭。如果不想要这种行为，只需给open()提供一个可选的closefd=False参数即可。示例如下：

```
# Create a file object, but don't close underlying fd when done



f = open(fd, 'wt', closefd=False)
...
```

5.18.3 讨论

在UNIX系统上，这种包装文件描述符的技术

可以用来方便地对以不同方式打开的I/O通道（即，管道、socket等）提供一个类似于文件的接口。例如，下面是一个有关socket的例子：

```
from socket import


socket, AF_INET, SOCK_STREAM
def


echo_client(client_sock, addr):
    print


('Got connection from', addr)
    # Make text-mode file wrappers for socket reading/writing



    client_in = open(client_sock.fileno(), 'rt', encoding='latin-
                          closefd=False)
    client_out = open(client_sock.fileno(), 'wt', encoding='latin
                          closefd=False)
    # Echo lines back to the client using file I/O



    for


line in


client_in:
        client_out.write(line)
        client_out.flush()
    client_sock.close()
def
```

```
echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while


True:
        client, addr = sock.accept()
        echo_client(client, addr)
```

需要重点强调的是，上面的例子仅仅只是用来说明内建的open()函数的一种特性，而且只能工作在基于UNIX的系统之上。如果想在socket上加上一个类似文件的接口，并且需要做到跨平台，那么就应该使用socket的makefile()方法来替代。但是，如果不需要考虑可移植性的话，就会发现上面给出的解决方案在性能上要比makefile()高出不少。

也可以利用这项技术为一个已经打开的文件创建一种别名，使得它的工作方式能够稍微区别于首次打开时的样子。比方说，下面这段代码告诉我们如何创建一个文件对象，使得它能够在stdout上产生出二进制数据（通常stdout是以文本模式打开的）：

```
import sys
```

```
# Create a binary-mode file for stdout



bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n


')
bstdout.flush()
```

尽管我们可以将一个已存在的文件描述符包装成一个合适的文件，但是请注意，并非所有的文件模式都可以得到支持，而且某些特定类型的文件描述符可能还带有有趣的副作用（尤其是在面对错误处理、文件结尾的情况时）。具体的行为也可能因为操作系统的不同而有所区别。特别是，上面所有的示例代码都没法在非UNIX系统上工作。因此，最基本的底线就是需要对自己的实现进行彻底的测试，确保代码能够按照期望的方式工作。

5.19 创建临时文件和目录

5.19.1 问题

当程序运行时，我们需要创建临时文件或目录以便使用。在这之后，我们可能希望将这些文件和目录销毁掉。

5.19.2 解决方案

tempfile模块中有各种函数可以用来完成这个任务。要创建一个未命名的临时文件，可以使用tempfile.TemporaryFile:

```
from tempfile import


TemporaryFile
with


TemporaryFile('w+t') as


f:
     # Read/write to the file



     f.write('Hello World\n
```

```
')
     f.write('Testing\n

')
     # Seek back to beginning and read the data



     f.seek(0)
     data = f.read()
# Temporary file is destroyed
```

或者如果你喜欢的话，也可以像这样使用文件：

```
f = TemporaryFile('w+t')
# Use the temporary file




...
f.close()
# File is destroyed
```

TemporaryFile()的第一个参数是文件模式，通常以w+t处理文本模式而以w+b处理二进制数据。这个模式可同时支持读写，在这里是很有用的，因为关闭文件后再来修改模式实际上会销毁文件对象。此外，TemporaryFile()也可以接受和内建的open()函数一样的参数。示例如下：

```
with

TemporaryFile('w+t', encoding='utf-8', errors='ignore') as

f:
    ...
```

在大多数UNIX系统上，由TemporaryFile()创建的文件都是未命名的，而且在目录中也没有对应的条目。如果想解放这种限制，可以使用NamedTemporaryFile()来替代。示例如下：

```
from tempfile import

NamedTemporaryFile
with

NamedTemporaryFile('w+t') as

f:
    print

('filename is:', f.name)


...
# File automatically destroyed
```

这里，在已打开文件的f.name属性中就包含了临时文件的文件名。如果需要将它传给其他需要打开这个文件的代码时，这就显得很有用了。对于TemporaryFile()而言，结果文件会在关闭时自动删除。如果不想要这种行为，可以提供一个delete=False关键字参数。示例如下：

```
with NamedTemporaryFile('w+t', delete=False) as f:
    print('filename is:', f.name)
...
```

要创建一个临时目录，可以使用tempfile.TemporaryDirectory()来实现。示例如下：

```
from tempfile import

TemporaryDirectory
with

TemporaryDirectory() as

dirname:
     print

('dirname is:', dirname)


# Use the directory




...
# Directory and all contents destroyed
```

5.19.3　讨论

要和临时文件还有临时目录打交道，最方便的方式就是使用TemporaryFile()、NamedTemporaryFile()以及TemporaryDirectory()这

三个函数了。因为它们能自动处理有关创建和清除的所有步骤。从较低的层次来看，也可以使用mkstemp()和mkdtemp()来创建临时文件和目录。示例如下：

```
>>> import tempfile



>>> tempfile.mkstemp()
(3, '/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp7fefhv
>>> tempfile.mkdtemp()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp5wvcv6'
>>>
```

但是，这些函数并不会进一步去处理文件管理的任务。例如，mkstemp()函数只是简单地返回一个原始的操作系统文件描述符，然后由我们自行将其转换为一个合适的文件。同样地，如果想将文件清理掉的话，这个任务也是由我们自己完成。

一般情况下，临时文件都是在系统默认的区域中创建的，比如/var/tmp或者类似的地方。要找出实际的位置，可以使用tempfile.gettempdir()函数。示例如下：

```
>>> tempfile.gettempdir()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-'
>>>
```

所有同临时文件相关的函数都允许使用prefix、suffix和dir关键字参数来覆盖目录。例如：

```
>>> f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/
>>> f.name
'/tmp/mytemp8ee899.txt'
>>>
```

最后但也很重要的是，在可能的范围内，tempfile模块创建的临时文件都是以最安全的方式来进行的。这包括只为当前用户提供可访问的权限，并且在创建文件时采取了相应的步骤来避免出现竞态条件。请注意，在不同的平台下这可能会有一些区别。因此，对于更精细的要点，应该确保自己去查阅官方文档（http://docs.python.org/3/library/tempfile.html ）。

5.20 同串口进行通信

5.20.1 问题

我们想通过串口读取和写入数据，典型情况下是同某种硬件设备进行交互（即，机器人或传感器）。

5.20.2 解决方案

尽管可以直接通过Python内建的I/O原语来完成这个任务，但对于串口通信来说，最好还是使用pySerial包比较好。这个包使用起来非常简单，要打开一个串口，只要使用这样的代码即可：

```
import serial

ser = serial.Serial('/dev/tty.usbmodem641', # Device name varies

                     baudrate=9600,
                     bytesize=8,
                     parity='N',
                     stopbits=1)
```

设备名称可能会根据设备的类型和操作系统而有所不同。比如，在Windows上，可以使用0、1这样的数字代表设备来打开通信端口，比如“COM0”和“COM1”。一旦打开后，就可以通过read()、readline()和write()调用来读写数据了。示例如下：

```
ser.write(b'G1 X50 Y50\r\n


')
resp = ser.readline()
```

从这一点来看，大部分情况下的串口通信任务应该是非常简单的。

5.20.3　讨论

尽管表面上看起来很简单，串口通信常常会变得相当混乱。应该使用一个像pySerial这样的包的原因就在于它对一些高级特性提供了支持（即，超时处理、流控、刷新缓冲区、握手机制等）。比如，如果想开启RTS-CTS握手，只要简单地为Serial()提供一个rtscts=True关键字参数即可。pySerial提供的文档非常棒，所以在这里多解释也

没多大用处。

请记住所有涉及串口的I/O操作都是二进制的。因此，确保在代码中使用的是字节而不是文本（或者根据需要执行适当的文本编码/解码操作）。当需要创建以二进制编码的命令或者数据包时，struct模块也会起到不少作用。

5.21　序列化Python对象

5.21.1　问题

我们需要将Python对象序列化为字节流，这样就可以将其保存到文件中、存储到数据库中或者通过网络连接进行传输。

5.21.2　解决方案

序列化数据最常见的做法就是使用pickle模块。要将某个对象转储到文件中，可以这样做：

```
import pickle

data = ... # Some Python object

f = open('somefile', 'wb')
pickle.dump(data, f)
```

要将对象转储为字符串，可以使用pickle.dumps()：

```
s = pickle.dumps(data)
```

如果要从字节流中重新创建出对象，可以使用pickle.load()或者pickle.loads()函数。示例如下：

```
# Restore from a file



f = open('somefile', 'rb')
data = pickle.load(f)
# Restore from a string



data = pickle.loads(s)
```

5.21.3　讨论

对于大部分程序来说，只要掌握dump()和load()函数的用法就可以高效地利用pickle模块了。pickle模块能够兼容大部分Python数据类型和用户自定义的类实例。如果正在使用的库可以保存/恢复Python对象到数据库中，或者通过网络传输对象，那么很有可能就在使用pickle。

pickle是一种Python专有的自描述式的数据编码。说到自描述，因为序列化的数据中包含有每个

对象的开始和结束以及有关对象类型的信息。因此，不需要担心应该如何定义记录——pickle就能完成了。例如，如果需要处理多个对象，可以这么做：

```
>>> import pickle



>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

可以对函数、类以及实例进行pickle处理，但由此产生的数据只会对代码对象所关联的名称进行编码。例如：

```
>>> import math



>>> import pickle.
```

```
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>
```

当对数据做反序列化处理时，会假设所有所需的源文件都是可用的。模块、类以及函数会根据需要自动导入。对于需要在不同机器上的解释器之间共享Python数据的应用，这会成为一个潜在的维护性问题，因为所有的机器都必须能够访问到相同的源代码。



警告

绝对不能对非受信任的数据使用pickle.load()。由于会产生副作用，pickle会自动加载模块并创建实例。但是，了解pickle是如何运作的骇客可以故意创建出格式不正确的数据，使得Python解释器有机会去执行任意的系统命令。因此，有必要将pickle限制为只在内部使用，解释器和数据之间要能够彼此验证对方。

某些特定类型的对象是无法进行pickle操作的。这些对象一般来说都会涉及某种外部系统状态，比如打开的文件、打开的网络连接、线程、进

程、栈帧等。用户自定义的类有时候可以通过提供__getstate__()和__setstate__()方法来规避这些限制。如果定义了这些方法，pickle.dump()就会调用__getstate__()来得到一个可以被pickle处理的对象。同样地，在unpickle的时候就会调用__setstate__()了。为了说明，下面这个类在内部定义了一个线程，但是仍然可以进行pickle/unpickle操作：

```
# countdown.py

import time

import threading


class Countdown

:
    def

__init__(self, n):

self.n = n

self.thr = threading.Thread(target=self.run)
```

```
self.thr.daemon = True


self.thr.start()

def


run(self):
    while


self.n > 0:
        print


('T-minus', self.n)


self.n -= 1


time.sleep(5)

def


__getstate__(self):
    return


self.n

def


__setstate__(self, n):
```

```
self.__init__(n)
```

用下面的代码试验一下pickle操作：

```
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()
```

现在退出Python，重启之后再试试这个：

```
>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
```

```
T-minus 19
T-minus 18
...
```

可以看到线程又魔法般地重新焕发出生命了，而且是从上次执行pickle操作时剩下的计数开始执行。

对于大型的数据结构，比如由array模块或numpy库创建的二进制数组，pickle就不是一种特别高效的编码了。如果需要移动大量的数组型数据，那么最好简单地将数据按块保存在文件中，或者使用更加标准的编码，比如HDF5（由第三方库支持）。

由于pickle是Python的专有特性，而且同源代码的关联紧密，因此不应该把pickle作为长期存储的格式。比如说，如果源代码发生改变，那么存储的所有数据就会失效且变得无法读取。坦白说，要将数据保存在数据库和归档存储中，最好使用一种更加标准的数据编码，比如XML、CSV或者JSON。这些编码方式的标准化程度更高，许多编程语言都支持，而且更能适应于源代码的修改。

最后但同样重要的是，请注意pickle模块中有着大量的选项和棘手的阴暗角落。对于大部分常见的用途，我们不必担心这些问题。但是如果要构建一个大型的应用，其中要用pickle来做序列化的话，那么就应该好好参考官方文档（http://docs.python.org/3/library/pickle.html ）。

第6章　数据编码与处理

本章主要关注的重点是利用Python来处理以各种常见编码形式所呈现出的数据，比如CSV文件、JSON、XML以及二进制形式的打包记录。与数据结构那章不同，本章不会把重点放在特定的算法之上，而是着重处理数据在程序中的输入和输出问题上。

6.1　读写CSV数据

6.1.1　问题

我们想要读写CSV文件中的数据。

6.1.2　解决方案

对于大部分类型的CSV数据，我们都可以用csv库来处理。比如，假设在名为stocks.csv的文件中包含有如下的股票市场数据：

```
Symbol,Price,Date,Time,Change,Volume
"AA",39.48,"6/11/2007","9:36am",-0.18,181800
"AIG",71.38,"6/11/2007","9:36am",-0.15,195500
"AXP",62.58,"6/11/2007","9:36am",-0.46,935000
"BA",98.31,"6/11/2007","9:36am",+0.12,104800
"C",53.08,"6/11/2007","9:36am",-0.25,360900
"CAT",78.29,"6/11/2007","9:36am",-0.23,225400
```

下面的代码示例告诉我们如何将这些数据读取为元组序列：

```
import csv
```

```
with


open('stocks.csv') as


f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for


row in


f_csv:
        # Process row



        ...
```

在上面的代码中，row将会是一个元组。因此，要访问特定的字段就需要用到索引，比如row[0]（表示Symbol）和row[4]（表示Change）。

由于这样的索引常常容易混淆，因此这里可以考虑使用命名元组。示例如下：

```
from collections import


namedtuple
with
```

```
open('stock.csv') as


f:
    f_csv = csv.reader(f)
    headings = next(f_csv)
    Row = namedtuple('Row', headings)
    for


r in


f_csv:
        row = Row(*r)
        # Process row



        ...
```

这样就可以使用每一列的标头比如row.Symbol和row.Change来取代之前的索引了。应该要指出的是，这个方法只有在每一列的标头都是合法的Python标识符时才起作用。如果不是的话，就必须调整原始的标头（比如，把非标识符字符用下划线或其他类似的符号取代）。

另一种可行的方式是将数据读取为字典序列。可以用下面的代码实现：

```
import csv
```

```
with

open('stocks.csv') as

f:
    f_csv = csv.DictReader(f)
    for

row in

f_csv:
        # process row

        ...
```

在这个版本中，可以通过行标头来访问每行中的元素。比如，row['Symbol']或者row['Change']。

要写入CSV数据，也可以使用csv模块来完成，但是要创建一个写入对象。示例如下：

```
headers = ['Symbol','Price','Date','Time','Change','Volume']
rows = [('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
        ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
        ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000),
       ]
with

open('stocks.csv','w') as
```

```
f:
    f_csv = csv.writer(f)
    f_csv.writerow(headers)
    f_csv.writerows(rows)
```

如果数据是字典序列，那么可以这样处理：

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [{'Symbol':'AA', 'Price':39.48, 'Date':'6/11/2007',
          'Time':'9:36am', 'Change':-0.18, 'Volume':181800},
        {'Symbol':'AIG', 'Price': 71.38, 'Date':'6/11/2007',
          'Time':'9:36am', 'Change':-0.15, 'Volume': 195500},
        {'Symbol':'AXP', 'Price': 62.58, 'Date':'6/11/2007',
          'Time':'9:36am', 'Change':-0.46, 'Volume': 935000},
        ]
with


open('stocks.csv','w') as


f:
    f_csv = csv.DictWriter(f, headers)
    f_csv.writeheader()
    f_csv.writerows(rows)
```

6.1.3 讨论

应该总是选择使用csv模块来处理，而不是自己手动分解和解析CSV数据。比如，我们可能会倾向于写出这样的代码：

```
with

open('stocks.csv') as

f:
    for

line in

f:
        row = line.split(',')
        # process row

        ...
```

这种方式的问题在于仍然需要自己处理一些令人厌烦的细节问题。比如说，如果有任何字段是被引号括起来的，那么就要自己去除引号。此外，如果被引用的字段中恰好包含有一个逗号，那么产生出的那一行会因为大小错误而使得代码崩溃（因为原始数据也是用逗号分隔的）。

默认情况下，csv库被实现为能够识别微软Excel所采用的CSV编码规则。这也许是最为常见的CSV编码规则了，能够带来最佳的兼容性。但是，如果查阅csv的文档，就会发现有几种方法可以将

编码微调为其他的格式（例如，修改分隔字符等）。比方说，如果想读取以tab键分隔的数据，可以使用下面的代码：

```
# Example of reading tab-separated values

with

open('stock.tsv') as

f:
    f_tsv = csv.reader(f, delimiter='\t

')
    for

row in

f_tsv:
        # Process row

        ...
```

如果正在读取CSV数据并将其转换为命名元组，那么在验证列标题时要小心。比如，某个CSV文件中可能在标题行中包含有非法的标识符字符，

就像下面的示例这样[1]：

```
Street Address,Num-Premises,Latitude,Longitude
5412 N CLARK,10,41.980262,-87.668452
```

这会使得创建命名元组的代码出现ValueError异常。要解决这个问题，应该首先整理标题。例如，可以对非法的标识符字符进行正则替换，示例如下：

```
import re

with

open('stock.csv') as

f:
    f_csv = csv.reader(f)
    headers = [ re.sub('[^a-zA-Z_]', '_', h) for

h in

next(f_csv) ]
    Row = namedtuple('Row', headers)
    for

r in
```

```
f_csv:
        row = Row(*r)
        # Process row



        ...
```

此外，还需要重点强调的是，csv模块不会尝试去解释数据或者将数据转换为除字符串之外的类型。如果这样的转换很重要，那么这就是我们需要自行处理的问题。下面这个例子演示了对CSV数据进行额外的类型转换：

```
col_types = [str, float, str, str, float, int]
with


open('stocks.csv') as


f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for


row in


f_csv:
        # Apply conversions to the row items



        row = tuple(convert(value) for
```

```
convert, value in


zip(col_types, row))
        ...
```

作为另外一种选择，下面这个例子演示了将选中的字段转换为字典：

```
print


('Reading as dicts with type conversion')
field_types = [ ('Price', float),
                ('Change', float),
                ('Volume', int) ]
with


open('stocks.csv') as


f:
    for


row in


csv.DictReader(f):
        row.update((key, conversion(row[key]))
                    for


key, conversion in
```

```
field_types)
        print


(row)
```

一般来说，对于这样的转换都应该小心为上。在现实世界中，CSV文件可能会缺少某些值，或者数据损坏了，以及出现其他一些可能会使类型转换操作失败的情况，这都是很常见的。因此，除非可以保证数据不会出错，否则就需要考虑这些情况（也许需要加上适当的异常处理代码）。

最后，如果我们的目标是通过读取CSV数据来进行数据分析和统计，那么应该看看Pandas这个Python包（http://pandas.pydata.org ））。Pandas中有一个方便的函数pandas.read_csv()，能够将CSV数据加载到DataFrame对象中。之后，就可以生成各种各样的统计摘要了，还可以对数据进行筛选并执行其他类型的高级操作。6.13节中给出了一个这样的例子。

6.2　读写JSON数据

6.2.1　问题

我们想读写以JSON（JavaScript Object Notation）格式编码的数据。

6.2.2　解决方案

json模块中提供了一种简单的方法来编码和解码JSON格式的数据。这两个主要的函数就是json.dumps()以及json.loads()。这两个函数在命名上借鉴了其他序列化处理库的接口，比如pickle。下面的示例展示了如何将Python数据结构转换为JSON：

```
import json



data = {
    'name' : 'ACME',
    'shares' : 100,
    'price' : 542.23
}
json_str = json.dumps(data)
```

而接下来的示例告诉我们如何把JSON编码的字符串再转换回Python数据结构：

```
data = json.loads(json_str)
```

如果要同文件而不是字符串打交道的话，可以选择使用json.dump()以及json.load()来编码和解码JSON数据。示例如下：

```
# Writing JSON data

with

open('data.json', 'w') as

f:
     json.dump(data, f)
# Reading data back

with

open('data.json', 'r') as

f:
     data = json.load(f)
```

6.2.3 讨论

JSON编码支持的基本类型有None、bool、int、float和str，当然还有包含了这些基本类型的列表、元组以及字典。对于字典，JSON会假设键（key）是字符串（字典中的任何非字符串键都会在编码时转换为字符串）。要符合JSON规范，应该只对Python列表和字典进行编码。此外，在Web应用中，把最顶层对象定义为字典是一种标准做法。

JSON编码的格式几乎与Python语法一致，只有几个小地方稍有不同。比如，True会被映射为true，False会被映射为false，而None会被映射为null。下面的示例展示了编码看起来是怎样的：

```
>>> json.dumps(False)
'false'
>>> d = {'a': True,
...


'b': 'Hello',
...


'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

如果要检查从JSON中解码得到的数据，那么仅仅将其打印出来就想确定数据的结构通常是比较困难的——尤其是如果数据中包含了深层次的嵌套结构或者有许多字段时。为了帮助解决这个问题，考虑使用pprint模块中的pprint()函数。这么做会把键按照字母顺序排列，并且将字典以更加合理的方式进行输出。下面的示例展示了应该如何对Twitter上的搜索结果以漂亮的格式进行输出：

```
>>> from urllib.request import


urlopen
>>> import json



>>> u = urlopen('http://search.twitter.com/search.json?q=python&r
>>> resp = json.loads(u.read().decode('utf-8'))
>>> from pprint import


pprint
>>> pprint(resp)
{'completed_in': 0.074,
 'max_id': 264043230692245504,
 'max_id_str': '264043230692245504',
 'next_page': '?page=2&max_id=264043230692245504&q=python&rpp=5',
 'page': 1,
 'query': 'python',
 'refresh_url': '?since_id=264043230692245504&q=python',
 'results': [{'created_at': 'Thu, 01 Nov 2012 16:36:26 +0000',
              'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:14 +0000',
              'from_user': ...
            },
            {'created_at': 'Thu, 01 Nov 2012 16:36:13 +0000',
```

```
              'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:07 +0000',
              'from_user': ...
             }
             {'created_at': 'Thu, 01 Nov 2012 16:36:04 +0000',
              'from_user': ...
             }],
 'results_per_page': 5,
 'since_id': 0,
 'since_id_str': '0'}
>>>
```

一般来说，JSON解码时会从所提供的数据中创建出字典或者列表。如果想创建其他类型的对象，可以为json.loads()方法提供object_pairs_hook或者object_hook参数。例如，下面的示例展示了我们应该如何将JSON数据解码为OrderedDict（有序字典），这样可以保持数据的顺序不变：

```
>>> s = '{"name": "ACME", "shares": 50, "price": 490.1}'
>>> from collections import


OrderedDict
>>> data = json.loads(s, object_pairs_hook=OrderedDict)
>>> data
OrderedDict([('name', 'ACME'), ('shares', 50), ('price', 490.1)])
>>>
```

而下面的代码将JSON字典转变为Python对象：

```
>>> class JSONObject


:
...     def


__init__(self, d):
...


self.__dict__ = d
...




>>>
>>> data = json.loads(s, object_hook=JSONObject)
>>> data.name
'ACME'
>>> data.shares
50
>>> data.price
490.1
>>>
```

在上一个示例中，通过解码JSON数据而创建的字典作为单独的参数传递给了__init__()。之后就可以自由地根据需要来使用它了，比如直接将它当做对象的字典实例来用。

有几个选项对于编码JSON来说是很有用的。如果想让输出格式变得漂亮一些，可以在json.dumps()函数中使用indent参数。这会使得数据

能够像pprint()函数那样以漂亮的格式打印出来。示例如下：

```
>>> print


(json.dumps(data))
{"price": 542.23, "name": "ACME", "shares": 100}
>>> print


(json.dumps(data, indent=4))
{
    "price": 542.23,
    "name": "ACME",
    "shares": 100
}
>>>
```

如果想在输出中对键进行排序处理，可以使用sort_keys参数：

```
>>> print


(json.dumps(data, sort_keys=True))
{"name": "ACME", "price": 542.23, "shares": 100}
>>>
```

类实例一般是无法序列化为JSON的。比如说：

```
>>> class Point


:
...     def


__init__(self, x, y):
...


self.x = x
...


self.y = y
...



>>> p = Point(2, 3)
>>> json.dumps(p)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/json/__init__.py", line 226, in
    return


_default_encoder.encode(obj)
  File "/usr/local/lib/python3.3/json/encoder.py", line 187, in e
    chunks = self.iterencode(o, _one_shot=True)
  File "/usr/local/lib/python3.3/json/encoder.py", line 245, in i
    return


_iterencode(o, 0)
  File "/usr/local/lib/python3.3/json/encoder.py", line 169, in d
    raise TypeError


(repr(o) + " is not JSON serializable")
TypeError: <__main__.Point object at 0x1006f2650> is not JSON ser
>>>
```

如果想序列化类实例，可以提供一个函数将类实例作为输入并返回一个可以被序列化处理的字典。示例如下：

```
def

serialize_instance(obj):
    d = { '__classname__' : type(obj).__name__ }
    d.update(vars(obj))
    return d
```

如果想取回一个实例，可以编写这样的代码来处理：

```
# Dictionary mapping names to known classes

classes = {
    'Point' : Point
}
def

unserialize_object(d):
    clsname = d.pop('__classname__', None)
    if

clsname:
```

```
        cls = classes[clsname]
        obj = cls.__new__(cls) # Make instance without calling __



        for


key, value in


d.items():
            setattr(obj, key, value)
            return


obj
    else


:
        return


d
```

最后给出如何使用这些函数的示例：

```
>>> p = Point(2,3)
>>> s = json.dumps(p, default=serialize_instance)
>>> s
'{"__classname__": "Point", "y": 3, "x": 2}'
>>> a = json.loads(s, object_hook=unserialize_object)
>>> a
<__main__.Point object at 0x1017577d0>
>>> a.x
2
>>> a.y
3
>>>
```

json模块中还有许多其他的选项，这些选项可用来控制对数字、特殊值（比如NaN）等的底层解释行为。请参阅文档（http://docs.python.org/3/library/json.html ）以获得进一步的细节。

6.3 解析简单的XML文档

6.3.1 问题

我们想从一个简单的XML文档中提取出数据。

6.3.2 解决方案

xml.etree.ElementTree模块可用来从简单的XML文档中提取出数据。为了说明，假设想对Planet Python（http://planet.python.org ）上的RSS订阅做解析并生成一个总结报告。下面的脚本可以完成这个任务：

```
from urllib.request import


urlopen
from xml.etree.ElementTree import


parse
# Download the RSS feed and parse it



u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)
# Extract and output tags of interest
```

```
for

item in

doc.iterfind('channel/item'):
    title = item.findtext('title')
    date = item.findtext('pubDate')
    link = item.findtext('link')

    print

(title)
    print

(date)
    print

(link)
    print

()
```

如果运行上面的脚本，会得到类似这样的输出：

```
Steve Holden: Python for Data Analysis
Mon, 19 Nov 2012 02:13:51 +0000
http://holdenweb.blogspot.com/2012/11/python-for-data-analysis.ht
Vasudev Ram: The Python Data model (for v2 and v3)
Sun, 18 Nov 2012 22:06:47 +0000
http://jugad2.blogspot.com/2012/11/the-python-data-model.html
Python Diary: Been playing around with Object Databases
```

```
Sun, 18 Nov 2012 20:40:29 +0000
http://www.pythondiary.com/blog/Nov.18,2012/been-...-object-datal
Vasudev Ram: Wakari, Scientific Python in the cloud
Sun, 18 Nov 2012 20:19:41 +0000
http://jugad2.blogspot.com/2012/11/wakari-scientific-python-in-cl
Jesse Jiryu Davis: Toro: synchronization primitives for Tornado c
Sun, 18 Nov 2012 20:17:49 +0000
http://feedproxy.google.com/~r/EmptysquarePython/~3/_DOZT2Kd0hQ/
```

显然，如果想做更多的处理，就需要将print()函数替换为其他更加有趣的处理函数。

6.3.3 讨论

在许多应用中，同XML编码的数据打交道是很常见的事情。这不仅是因为XML作为一种数据交换格式在互联网中使用广泛，而且XML还是用来保存应用程序数据（例如文字处理、音乐库等）的常用格式。本节后面的讨论假设读者已经熟悉XML的基本概念。

在许多情况下，XML如果只是简单地用来保存数据，那么文档结构就是紧凑而直接的。例如，上面示例中的RSS订阅源看起来类似于如下的XML文档：

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
<channel>
  <title>Planet Python</title>
```

```
  <link>http://planet.python.org/</link>
  <language>en</language>
  <description>Planet Python - http://planet.python.org/</descrip
  <item>
    <title>Steve Holden: Python for Data Analysis</title>
      <guid>http://holdenweb.blogspot.com/...-data-analysis.html
      <link>http://holdenweb.blogspot.com/...-data-analysis.html
      <description>...</description>
      <pubDate>Mon, 19 Nov 2012 02:13:51 +0000</pubDate>
  </item>
  <item>
    <title>Vasudev Ram: The Python Data model (for v2 and v3)</ti
    <guid>http://jugad2.blogspot.com/...-data-model.html</guid>
    <link>http://jugad2.blogspot.com/...-data-model.html</link>
    <description>...</description>
    <pubDate>Sun, 18 Nov 2012 22:06:47 +0000</pubDate>
    </item>
  <item>
    <title>Python Diary: Been playing around with Object Database
    <guid>http://www.pythondiary.com/...-object-databases.html</
    <link>http://www.pythondiary.com/...-object-databases.html</
    <description>...</description>
    <pubDate>Sun, 18 Nov 2012 20:40:29 +0000</pubDate>
  </item>
    ...
</channel>
</rss>
```

xml.etree.ElementTree.parse()函数将整个XML文档解析为一个文档对象。之后，就可以利用find()、iterfind()以及findtext()方法查询特定的XML元素。这些函数的参数就是特定的标签名称，比如channel/item或者title。

当指定标签时，需要整体考虑文档的结构。每一个查找操作都是相对于一个起始元素来展开的。

同样地，提供给每个操作的标签名也是相对于起始元素的。在示例代码中，对doc.iterfind('channel/item')的调用会查找所有在“channel”元素之下的“item”元素。doc代表着文档的顶层（顶层“rss”元素）。之后对item.findtext()的调用就相对于已找到的“item”元素来展开。

每个由ElementTree模块所表示的元素都有一些重要的属性和方法，它们对解析操作十分有用。tag属性中包含了标签的名称，text属性中包含有附着的文本，而get()方法可以用来提取出属性（如果有的话）。示例如下：

```
>>> doc
<xml.etree.ElementTree.ElementTree object at 0x101339510>
>>> e = doc.find('channel/title')
>>> e
<Element 'title' at 0x10135b310>
>>> e.tag
'title'
>>> e.text
'Planet Python'
>>> e.get('some_attribute')
>>>
```

应该要指出的是xml.etree.ElementTree并不是解析XML的唯一选择。对于更加高级的应用，应该考虑使用lxml。lxml采用的编程接口和ElementTree一样，因此本节中展示的示例能够以同样的方式用

lxml实现。只需要将第一个导入语句修改为from lxml.etree import parse即可。lxml完全兼容于XML标准，这为我们提供了极大的好处。此外，lxml运行起来非常快速，还提供验证、XSLT以及XPath这样的功能支持。

6.4　以增量方式解析大型XML文件

6.4.1　问题

我们需要从一个大型的XML文档中提取出数据，而且对内存的使用要尽可能少。

6.4.2　解决方案

任何时候，当要面对以增量方式处理数据的问题时，都应该考虑使用迭代器和生成器。下面是一个简单的函数，可用来以增量方式处理大型的XML文件，它只用到了很少量的内存：

```
from xml.etree.ElementTree import


iterparse
def


parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Skip the root element
```

```
    next(doc)
    tag_stack = []
    elem_stack = []
    for

event, elem in

doc:
        if

event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif

event == 'end':
            if

tag_stack == path_parts:
                yield

elem
                elem_stack[-2].remove(elem)
            try

:
                tag_stack.pop()
                elem_stack.pop()
            except IndexError

:
                pass
```

要测试这个函数，只需要找一个大型的XML文件来配合测试即可。这种大型的XML文件常常可以在政府以及数据公开的网站上找到。比如，可以下载芝加哥的坑洞数据库XML。在写作本书时，这个下载文件中有超过100000行的数据，它们按照如下的方式编码：

```
<response>
  <row>
    <row ...>
      <creation_date>2012-11-18T00:00:00</creation_date>
      <status>Completed</status>
      <completion_date>2012-11-18T00:00:00</completion_date>
      <service_request_number>12-01906549</service_request_number
      <type_of_service_request>Pot Hole in Street</type_of_servic
      <current_activity>Final Outcome</current_activity>
      <most_recent_action>CDOT Street Cut ... Outcome</most_recen
      <street_address>4714 S TALMAN AVE</street_address>
      <zip>60632</zip>
      <x_coordinate>1159494.68618856</x_coordinate>
      <y_coordinate>1873313.83503384</y_coordinate>
      <ward>14</ward>
      <police_district>9</police_district>
      <community_area>58</community_area>
      <latitude>41.808090232127896</latitude>
      <longitude>-87.69053684711305</longitude>
      <location latitude="41.808090232127896"
                        longitude="-87.69053684711305" />
    </row>
    <row ...>
      <creation_date>2012-11-18T00:00:00</creation_date>
      <status>Completed</status>
      <completion_date>2012-11-18T00:00:00</completion_date>
      <service_request_number>12-01906695</service_request_number
      <type_of_service_request>Pot Hole in Street</type_of_servic
      <current_activity>Final Outcome</current_activity>
      <most_recent_action>CDOT Street Cut ... Outcome</most_recen
```

```
      <street_address>3510 W NORTH AVE</street_address>
      <zip>60647</zip>
      <x_coordinate>1152732.14127696</x_coordinate>
      <y_coordinate>1910409.38979075</y_coordinate>
      <ward>26</ward>
      <police_district>14</police_district>
      <community_area>23</community_area>
      <latitude>41.91002084292946</latitude>
      <longitude>-87.71435952353961</longitude>
      <location latitude="41.91002084292946"
                       longitude="-87.71435952353961" />
    </row>
  </row>
</response>
```

假设我们想编写一个脚本来根据坑洞的数量对邮政编码（ZIP code）进行排序。可以编写如下的代码来实现：

```
from xml.etree.ElementTree import


parse
from collections import


Counter
potholes_by_zip = Counter()
doc = parse('potholes.xml')
for


pothole in


doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1
for
```

```
zipcode, num in


potholes_by_zip.most_common():
    print


(zipcode, num)
```

这个脚本存在的唯一问题就是它将整个XML文件都读取到内存中后再做解析。在我们的机器上，运行这个脚本需要占据450 MB内存。但是如果使用下面这份代码，程序只做了微小的修改：

```
from collections import


Counter
potholes_by_zip = Counter()
data = parse_and_remove('potholes.xml', 'row/row')
for


pothole in


data:
    potholes_by_zip[pothole.findtext('zip')] += 1
for


zipcode, num in
```

```
potholes_by_zip.most_common():
    print


(zipcode, num)
```

这个版本的代码运行起来只用了7 MB内存——多么惊人的提升啊！

6.4.3　讨论

本节中的示例依赖于ElementTree模块中的两个核心功能。首先，iterparse()方法允许我们对XML文档做增量式的处理。要使用它，只需提供文件名以及一个事件列表即可。事件列表由1个或多个start/end，start-ns/end-ns组成。iterparse()创建出的迭代器产生出形式为（event，elem）的元组，这里的event是列出的事件，而elem是对应的XML元素。示例如下：

```
>>> data = iterparse('potholes.xml',('start','end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
```

```
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
>>>
```

当某个元素首次被创建但是还没有填入任何其他数据时（比如子元素），会产生start事件，而end事件会在元素已经完成时产生。尽管没有在本节示例中出现，start-ns和end-ns事件是用来处理XML命名空间声明的。

在这个示例中，start和end事件是用来管理元素和标签栈的。这里的栈代表着文档结构中被解析的当前层次（current hierarchical），同时也用来判断元素是否匹配传递给parse_and_remove()函数的请求路径。如果有匹配满足，就通过yield将其发送给调用者。

紧跟在yield之后的语句就是使得ElementTree能够高效利用内存的关键所在：

```
elem_stack[-2].remove(elem)
```

这一行代码使得之前通过yield产生出的元素从它们的父节点中移除。因此可假设其再也没有任何

其他的引用存在，因此该元素被销毁进而可以回收它所占用的内存。

这种迭代式的解析以及对节点的移除使得对整个文档的增量式扫描变得非常高效。在任何时刻都能构造出一棵完整的文档树。然而，我们仍然可以编写代码以直接的方式来处理XML数据。

这种技术的主要缺点就是运行时的性能。当进行测试时，将整个文档先读入内存的版本运行起来大约比增量式处理的版本快2倍。但是在内存的使用上，先读入内存的版本占用的内存量是增量式处理的60倍多。因此，如果内存使用量是更加需要关注的因素，那么显然增量式处理的版本才是大赢家。

6.5　将字典转换为XML

6.5.1　问题

我们想将Python字典中的数据转换为XML。

6.5.2　解决方案

尽管xml.etree.ElementTree库通常用来解析XML文档，但它同样也可以用来创建XML文档。例如，考虑下面这个函数：

```
from xml.etree.ElementTree import

Element
def

dict_to_xml(tag, d):
    '''

    Turn a simple dict of key/value pairs into XML

    '''
```

```
    elem = Element(tag)
    for


key, val in


d.items():
        child = Element(key)
        child.text = str(val)
        elem.append(child)
    return


elem
```

下面是使用这个函数的示例：

```
>>> s = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
>>> e = dict_to_xml('stock', s)
>>> e
<Element 'stock' at 0x1004b64c8>
>>>
```

转换的结果是一个Element实例。对于I/O操作来说，可以利用xml.etree.ElementTree中的tostring()函数将其转换为字节串。示例如下：

```
>>> from xml.etree.ElementTree import


tostring
>>> tostring(e)
b'<stock><price>490.1</price><shares>100</shares><name>GOOG</name
```

```
>>>
```

如果想为元素附加上属性，可以使用set()方法实现：

```
>>> e.set('_id','1234')
>>> tostring(e)
b'<stock _id="1234"><price>490.1</price><shares>100</shares><name
</stock>'
>>>
```

如果需要考虑元素间的顺序，可以创建OrderedDict（有序字典）来取代普通的字典。参见1.7节中对有序字典的介绍。

6.5.3 讨论

当创建XML时，也许会倾向于只使用字符串来完成。比如：

```
def

dict_to_xml_str(tag, d):
    '''
```

```
    Turn a simple dict of key/value pairs into XML


    '''



    parts = ['<{}>'.format(tag)]
    for


key, val in


d.items():
        parts.append('<{0}>{1}</{0}>'.format(key,val))
    parts.append('</{}>'.format(tag))
    return


''.join(parts)
```

问题在于如果尝试手工处理的话，那么这就是在自找麻烦。比如，如果字典中包含有特殊字符时会发生什么？

```
>>> d = { 'name' : '<spam>' }
>>> # String creation



>>> dict_to_xml_str('item',d)
'<item><name><spam></name></item>'
>>> # Proper XML creation
```

```
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name><spam></name></item>'
>>>
```

请注意在上面这个示例中，字符<和>分别被<和>取代了。

下面的提示仅供参考。如果需要手工对这些字符做转义处理，可以使用xml.sax.saxutils中的escape()和unescape()函数。示例如下：

```
>>> from xml.sax.saxutils import


escape, unescape
>>> escape('<spam>')
'<spam>'
>>> unescape(_)
'<spam>'
>>>
```

为什么说创建Element实例要比使用字符串好？除了可以产生出正确的输出外，其他的原因在于这样可以更加方便地将Element实例组合在一起，创建出更大的XML文档。得到的Element实例也能够以各种方式进行处理，完全不必担心解析XML文本方面的问题。最重要的是，我们能够站在

更高的层面上对数据进行各种处理，只在最后把结果作为字符串输出即可。

6.6 解析、修改和重写XML

6.6.1 问题

我们想读取一个XML文档，对它做些修改后再以XML的方式写回。

6.6.2 解决方案

xml.etree.ElementTree模块可以轻松完成这样的任务。从本质上来说，开始时可以按照通常的方式来解析文档。例如，假设有一个名为pred.xml的文档，它看起来是这样的：

```
<?xml version="1.0"?>
<stop>



    <id>


14791</id>



    <nm>


Clark &
```

```
Balmoral</nm>
    <sri>
        <rt>
22</rt>
        <d>
North Bound</d>
        <dd>
North Bound</dd>
    </sri>
    <cr>
22</cr>
    <pre>
        <pt>
```

```
5 MIN</pt>

        <fd>
Howard</fd>

        <v>
1378</v>

        <rn>
22</rn>

    </pre>

    <pre>

        <pt>
15 MIN</pt>

        <fd>
```

```
Howard</fd>

        <v>

1867</v>

        <rn>

22</rn>

    </pre>

</stop>
```

下面的示例采用ElementTree来读取这个文档，并对文档的结构作出修改：

```
>>> from xml.etree.ElementTree import

parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>
>>> # Remove a few elements
```

```
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))
>>> # Insert a new element after <nm>...</nm>



>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'This is a test'
>>> root.insert(2, e)
>>> # Write back to a file



>>> doc.write('newpred.xml', xml_declaration=True)
>>>
```

这些操作的结果产生了一个新的XML文档，看起来是这样的：

```
<?xml version='1.0' encoding='us-ascii'?>
<stop>



    <id>


14791</id>



    <nm>
```

```
Clark &

Balmoral</nm>


    <spam>

This is a test</spam><pre>


        <pt>

5 MIN</pt>


        <fd>

Howard</fd>


        <v>

1378</v>


        <rn>

22</rn>


    </pre>
```

```
    <pre>
        <pt>
15 MIN</pt>
        <fd>
Howard</fd>
        <v>
1867</v>
        <rn>
22</rn>
    </pre>
</stop>
```

6.6.3 讨论

修改XML文档的结构是简单直接的，但是必须记住所有的修改主要是对父元素进行的，我们把它当做是一个列表一样对待。比如说，如果移除某个元素，那么就利用它的直接父节点的remove()方法完成。如果插入或添加新的元素，同样要使用父节点的insert()和append()方法来完成。这些元素也可以使用索引和切片操作来进行操控，比如element[i]或者是element[i:j]。

如果需要创建新的元素，可以使用Element类来完成，我们本节给出的示例中已经这么做了。这在6.5节中有更进一步的描述。

6.7 用命名空间来解析XML文档

6.7.1 问题

我们要解析一个XML文档，但是需要使用XML命名空间来完成。

6.7.2 解决方案

考虑使用了命名空间的如下XML文档：

```
<?xml version="1.0" encoding="utf-8"?>
<top>



  <author>


David Beazley</author>



  <content>



      <html


xmlns="http://www.w3.org/1999/xhtml">
```

```
          <head>

              <title>

Hello World</title>

          </head>

          <body>

              <h1>

Hello World!</h1>

          </body>

      </html>

  </content>

</top>
```

如果解析这个文档并尝试执行普通的查询操作，就会发现没那么容易实现，因为所有的东西都变得特别冗长啰嗦：

```
>>> # Some queries that work



>>> doc.findtext('author')
'David Beazley'
>>> doc.find('content')
<Element 'content' at 0x100776ec0>
>>> # A query involving a namespace (doesn't work)



>>> doc.find('content/html')
>>> # Works if fully qualified



>>> doc.find('content/{http://www.w3.org/1999/xhtml}html')
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> # Doesn't work



>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/head
>>> # Fully qualified



>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/'
...


'{http://www.w3.org/1999/xhtml}head/{http://www.w3.org/1999/xhtml
'Hello World'
>>>
```

通常可以将命名空间的处理包装到一个通用的类中，这样可以省去一些麻烦：

```
class XMLNamespaces

:
    def

__init__(self, **kwargs):


self.namespaces = {}
        for

name, uri in

kwargs.items():


self.register(name, uri)
    def

register(self, name, uri):


self.namespaces[name] = '{'+uri+'}'
    def

__call__(self, path):
        return

path.format_map(self.namespaces)
```

要使用这个类，可以按照下面的方式进行：

```
>>> ns = XMLNamespaces(html='http://www.w3.org/1999/xhtml')
>>> doc.find(ns('content/{html}html'))
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> doc.findtext(ns('content/{html}html/{html}head/{html}title'))
'Hello World'
>>>
```

6.7.3　讨论

对包含有命名空间的XML文档进行解析会非常繁琐。XMLNamespaces类的功能只是用来稍微简化一下这个过程，它允许在后序的操作中使用缩短的命名空间名称，而不必去使用完全限定的URI。

不幸的是，在基本的ElementTree解析器中不存在什么机制能获得有关命名空间的进一步信息。但是如果愿意使用iterparse()函数的话，还是可以获得一些有关正在处理的命名空间范围的信息。示例如下：

```
>>> from xml.etree.ElementTree import


iterparse
```

```
>>> for

evt, elem in

iterparse('ns2.xml', ('end', 'start-ns', 'end-ns')):
...     print

(evt, elem)
...



end <Element 'author' at 0x10110de10>
start-ns ('', 'http://www.w3.org/1999/xhtml')
end <Element '{http://www.w3.org/1999/xhtml}title' at 0x1011131b0
end <Element '{http://www.w3.org/1999/xhtml}head' at 0x1011130a8
end <Element '{http://www.w3.org/1999/xhtml}h1' at 0x101113310>
end <Element '{http://www.w3.org/1999/xhtml}body' at 0x101113260
end <Element '{http://www.w3.org/1999/xhtml}html' at 0x10110df70
end-ns None
end <Element 'content' at 0x10110de68>
end <Element 'top' at 0x10110dd60>
>>> elem # This is the topmost element



<Element 'top' at 0x10110dd60>
>>>
```

最后要提到的是，如果正在解析的文本用到了除命名空间之外的其他高级XML特性，那么最好还是使用lxml库。比方说，lxml对文档的DTD验证、更加完整的XPath支持和其他的高级XML特性提供了更好的支持。本节提到的技术只是为解析操作做

了一点修改，使得这个过程能够稍微简单一些。

6.8 同关系型数据库进行交互

6.8.1 问题

我们需要选择、插入或者删除关系型数据库中的行数据。

6.8.2 解决方案

在Python中，表达行数据的标准方式是采用元组序列。例如：

```
stocks = [



('GOOG', 100, 490.1),
    ('AAPL', 50, 545.75),
    ('FB', 150, 7.45),
    ('HPQ', 75, 33.2),
]
```

当数据以这种形式呈现时，通过Python标准的数据库API（在PEP 249中描述）来同关系型数据库进行交互相对来说就显得很直接了。该API的要点就是数据库上的所有操作都通过SQL查询来实现。

每一行输入或输出数据都由一个元组来表示。

为了说明，我们可以使用Python自带的sqlite3模块。如果正在使用一个不同的数据库（如MySQL、Postgres或者ODBC），就需要安装一个第三方的模块来支持。但是，底层的编程接口即使不完全相同的话也几乎是一致的。

第一步是连接数据库。一般来说，要调用一个connect()函数，提供类似数据库名称、主机名、用户名、密码这样的参数以及一些其他需要的细节。示例如下：

```
>>> import sqlite3



>>> db = sqlite3.connect('database.db')
>>>
```

要操作数据的话，下一步就要创建一个游标（cursor）。一旦有了游标，就可以开始执行SQL查询了。示例如下：

```
>>> c = db.cursor()
>>> c.execute('create table portfolio (symbol text, shares intege
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

要在数据中插入行序列，可以采用这样的语句：

```
>>> c.executemany('insert into portfolio values (?,?,?)', stocks
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

要执行查询操作，可以使用下面这样的语句：

```
>>> for

row in

db.execute('select * from portfolio'):
...     print

(row)
...



('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

如果想执行的查询操作需要接受用户提供的输入参数，请确保用?隔开参数，就像下面的示例这样：

```
>>> min_price = 100
>>> for


row in


db.execute('select * from portfolio where price >= ?',
                              (min_price,)):
...     print


(row)
...




('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

6.8.3　讨论

从较低的层次来看，同数据库的交互其实是一件非常直截了当的事。只要组成SQL语句然后将它们传给底层的模块就可以更新数据库或者取出数据了。尽管如此，这里还是有一些比较棘手的细节问题需要针对每种情况逐项考虑。

其中一个比较复杂的问题就是将数据库中的数据映射到Python的类型中。对于像日期这样的条目，最常见的是使用datetime模块中的datetime实例，或者也可能是time模块中用到的系统时间戳（system timestamps）。对于数值型的数据，尤其是涉及小数的金融数据，这些数字可以用decimal模块中的Decimal实例来表示。不幸的是，确切的映射关系会因数据库后端的不同而有所区别，因此必须去阅读相关的文档。

另一个极其重要的问题是需要考虑组成SQL语句的字符串。我们绝对不应该用Python的字符串格式化操作符（即%）或者.format()方法来创建这种字符串。如果给这样的格式化操作符提供的值是来自于用户的输入，那么这就等于将你的程序敞开大门迎接SQL注入攻击（参见http://xkcd.com/327 ）。在查询操作中，特殊的?通配符会指示数据库后端启用自己的字符串替换机制，这样才能做到安全（希望如此）。

可悲的是，数据库后端对通配符的支持并不一致。有许多模块采用的是?或%s，而其他一些可能会使用不同的符号，比如用:0或者:1来代表参数。这里再次说明，必须查阅正在使用的数据库模块的文档资料。数据库模块的paramstyle属性中也包含有关于引用样式的相关信息。

对于简单地三趸将数据从数据库表项中取出和输入，使用数据库API通常足够了。如果要处理更加复杂的任务，那么使用一种更高层次的接口就显得很有意义了，比如那些提供有对象关系映射组件（object-relational mapper，ORM）的接口。像SQLAlchemy（http://www.sqlalchemy.org ）这样的库允许数据库表项以Python类的形式来描述，在执行数据库操作时可隐藏大部分底层的SQL。

6.9 编码和解码十六进制数字

6.9.1 问题

我们需要将十六进制数组成的字符串解码为字节流，或者将字节流编码为十六进制数。

6.9.2 解决方案

如果需要编码或解码由十六进制数组成的原始字符串，可以使用binascii模块。示例如下：

```
>>> # Initial byte string



>>> s = b'hello'
>>> # Encode as hex



>>> import binascii



>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'
>>> # Decode back to bytes



>>> binascii.a2b_hex(h)
```

```
b'hello'
>>>
```

同样的功能也可以在base64模块中找到。示例如下：

```
>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656C6C6F'
>>> base64.b16decode(h)
b'hello'
>>>
```

6.9.3 讨论

对于大部分情况而言，采用上面给出的函数对十六进制数进行转换都是简单直接的。这两种技术的主要区别在于大写转换。base64.b16decode()和base64.b16encode()函数只能对大写形式的十六进制数进行操作，而binascii模块能够处理任意一种情况。

此外还需要重点提到的是编码函数产生的输出总是字节串。如果要将其强制转换为Unicode输

出，可能需要增加一些额外的解码操作。示例如下：

```
>>> h = base64.b16encode(s)
>>> print


(h)
b'68656C6C6F'
>>> print


(h.decode('ascii'))
68656C6C6F
>>>
```

当解码十六进制数时，b16decode()和a2b_hex()函数可接受字节串或Unicode字符串作为输入。但是，这些字符串中必须只能包含ASCII编码的十六进制数字。

6.10 Base64编码和解码

6.10.1 问题

我们需要采用Base64编码来对二进制数据做编码解码操作。

6.10.2 解决方案

base64模块中有两个函数——b64encode()和b64decode()——它们正是我们所需要的。示例如下：

```
>>> # Some byte data



>>> s = b'hello'
>>> import base64



>>> # Encode as Base64



>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='
>>> # Decode from Base64
```

```
>>> base64.b64decode(a)
b'hello'
>>>
```

6.10.3 讨论

Base64编码只能用在面向字节的数据上，比如字节串和字节数组。此外，编码过程的输出总是一个字节串。如果将Base64编码的数据同Unicode文本混在一起，那么可能需要执行一个额外的解码步骤。示例如下：

```
>>> a = base64.b64encode(s).decode('ascii')
>>> a
'aGVsbG8='
>>>
```

当解码Base64数据时，字节串和Unicode文本字符串都可以作为输入。但是，Unicode字符串中只能包含ASCII字符才行。

6.11　读写二进制结构的数组

6.11.1　问题

我们想将数据编码为统一结构的二进制数组，然后将这些数据读写到Python元组中去。

6.11.2　解决方案

要同二进制数据打交道的话，我们可以使用struct模块。下面的示例将一列Python元组写入到一个二进制文件中，通过struct模块将每个元组编码为一个结构。

```
from struct import


Struct
def


write_records(records, format, f):



'''
```

```
Write a sequence of tuples to a binary file of structures.

'''

record_struct = Struct(format)
    for

r in

records:

f.write(record_struct.pack(*r))

# Example

if

__name__ == '__main__':

records = [ (1, 2.3, 4.5),

(6, 7.8, 9.0),
```

```
(12, 13.4, 56.7) ]
with

open('data.b', 'wb') as

f:
     write_records(records, '<idd', f)
```

如果要将这个文件重新读取为一列Python元组的话，有好几种方法可以实现。首先，如果打算按块以增量式的方式读取文件的话，可以按照下面的示例来实现：

```
from struct import

Struct
def

read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda

: f.read(record_struct.size), b'')
    return

(record_struct.unpack(chunk) for
```

```
chunk in

chunks)
# Example

if

__name__ == '__main__':
    with

open('data.b','rb') as

f:
        for

rec in

read_records('<idd', f):
            # Process rec

...
```

如果只想用一个read()调用将文件全部读取到一个字节串中，然后再一块一块的做转换，那么可以编写如下的代码：

```
from struct import


Struct
def


unpack_records(format, data):
    record_struct = Struct(format)
    return


(record_struct.unpack_from(data, offset)
            for


offset in


range(0, len(data), record_struct.size))
# Example



if


__name__ == '__main__':
    with


open('data.b', 'rb') as


f:
        data = f.read()
    for


rec in


unpack_records('<idd', data):
        # Process rec
```

```
    ...
```

在这两种情况下得到的结果都是一个可迭代对象，它能够产生出之前保存在文件中的那些元组。

6.11.3 讨论

对于那些必须对二进制数据编码和解码的程序，我们常会用到struct模块。要定义一个新的结构，只要简单地创建一个Struct实例即可：

```
# Little endian 32-bit integer, two double precision floats

record_struct = Struct('<idd')
```

结构总是通过一组结构化代码来定义，比如i、d、f等（参见Python的文档http://docs.python.org/3/library/struct.html ）。这些代码同特定的二进制数据相对应，比如32位整数、64位浮点数、32位浮点数等。而第一个字符<

指定了字节序。在这个例子中表示为“小端序”。将字符修改为>就表示为大端序，或者用!来表示网络字节序。

得到的Struct实例有着多种属性和方法，它们可用来操纵那种类型的结构。size属性包含了以字节为单位的结构体大小，这对于I/O操作来说是很有用的。pack()和unpack()方法是用来打包和解包数据的。示例如下：

```
>>> from struct import


Struct
>>> record_struct = Struct('<idd')
>>> record_struct.size
20
>>> record_struct.pack(1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x
>>> record_struct.unpack(_)
(1, 2.0, 3.0)
>>>
```

有时候我们会发现pack()和unpack()会以模块级函数（module-level functions）的形式调用，就像下面的示例这样：

```
>>> import struct
```

```
>>> struct.pack('<idd', 1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x
>>> struct.unpack('<idd', _)
(1, 2.0, 3.0)
>>>
```

这么做行的通，但是比起创建一个单独的Struct实例来说还是显得不那么优雅，尤其是如果相同的结构会在代码中多处出现时。通过创建一个Struct实例，我们只用指定一次格式化代码，所有有用的操作都被漂亮地归组到了一起（通过实例方法来调用）。如果需要同结构打交道的话，这么做肯定会使得代码更容易维护（因为只需要修改一处即可）。

用来读取二进制结构的代码中涉及一些有趣而且优雅的编程惯用法（programming idioms）。在函数read_records()中，我们用iter()来创建一个迭代器，使其返回固定大小的数据块（参见5.8节）。这个迭代器会重复调用一个用户提供的可调用对象（即，lambda: f.read(record_struct.size)）直到它返回一个指定值为止（即，b''），此时迭代过程结束。示例如下：

```
>>> f = open('data.b', 'rb')
>>> chunks = iter(lambda
```

```
: f.read(20), b'')
>>> chunks
<callable_iterator object at 0x10069e6d0>
>>> for


chk in


chunks:
...     print


(chk)
...



b'\x01\x00\x00\x00ffffff\x02@\x00\x00\x00\x00\x00\x00\x12@'
b'\x06\x00\x00\x00333333\x1f@\x00\x00\x00\x00\x00\x00"@'
b'\x0c\x00\x00\x00\xcd\xcc\xcc\xcc\xcc\xcc*@\x9a\x99\x99\x99\x99
>>>
```

创建一个可迭代对象的原因之一在于这么做允许我们通过一个生成器表达式来创建records记录，就像解决方案中展示的那样。如果不采用这种方式，那么代码看起来就会像这样：

```
def


read_records(format, f):
    record_struct = Struct(format)
    while


True:
```

```
        chk = f.read(record_struct.size)
        if

chk == b'':
            break

        yield

record_struct.unpack(chk)
    return

records
```

在函数unpack_records()中我们采用了另一种方法。这里使用的unpack_from()方法对于从大型的二进制数组中提取出二进制数据是非常有用的，因为它不会创建任何临时对象或者执行内存拷贝动作。我们只需提供一个字节串（或者任意的数组），再加上一个字节偏移量，它就能直接从那个位置上将字段解包出来。

如果用的是unpack()而不是unpack_from()，那么需要修改代码，创建许多小的切片对象并且还要计算偏移量。示例如下：

```
def
```

```
unpack_records(format, data):
    record_struct = Struct(format)
    return


(record_struct.unpack(data[offset:offset + record_struct.size])
            for


offset in


range(0, len(data), record_struct.size))
```

这个版本的实现除了读取数据变得更加复杂之外，还需要完成许多工作，因为它得计算很多偏移量，拷贝数据，创建小的切片对象。如果打算从已读取的大型字节串中解包出许多结构的话，那么unpack_from()是更加优雅的方案。

我们可能会想在解包记录时利用collections模块中的namedtuple对象。这么做允许我们在返回的元组上设定属性名。示例如下：

```
from collections import


namedtuple
Record = namedtuple('Record', ['kind','x','y'])
with
```

```
open('data.p', 'rb') as


f:
    records = (Record(*r) for


r in


read_records('<idd', f))
for


r in


records:
    print


(r.kind, r.x, r.y)
```

如果正在编写一个需要同大量的二进制数据打交道的程序，最好使用像numpy这样的库。比如，与其将二进制数据读取到元组列表中，不如直接将数据读入到结构化的数组中，就像这样：

```
>>> import numpy as np



>>> f = open('data.b', 'rb')
>>> records = np.fromfile(f, dtype='<i,<d,<d')
>>> records
array([(1, 2.3, 4.5), (6, 7.8, 9.0), (12, 13.4, 56.7)],
      dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '<f8')])
```

```
>>> records[0]
(1, 2.3, 4.5)
>>> records[1]
(6, 7.8, 9.0)
>>>
```

最后但同样重要的是，如果面对的任务是从某种已知的文件结构中读取二进制数据（例如图像格式、shapefile、HDF5等），请先检查是否已有相应的Python模块可用。没必要的话就别去重复发明轮子了。

6.12 读取嵌套型和大小可变的二进制结构

6.12.1 问题

我们需要读取复杂的二进制编码数据，这些数据中包含有一系列嵌套的或者大小可变的记录。这种数据包括图片、视频、shapefile（zh.wikipedia.org/zh-cn/Shapefile）等。

6.12.2 解决方案

struct模块可用来编码和解码几乎任何类型的二进制数据结构。为了说明本节中提到的这种数据，假设我们有一个用Python数据结构表示的点的集合，这些点可用来组成一系列的三角形：

```
polys = [
          [ (1.0, 2.5), (3.5, 4.0), (2.5, 1.5) ],
          [ (7.0, 1.2), (5.1, 3.0), (0.5, 7.5), (0.8, 9.0) ],
          [ (3.4, 6.3), (1.2, 0.5), (4.6, 9.2) ],
        ]
```

现在假设要将这份数据编码为一个二进制文

件，这个文件的文件头可以表示为如下的形式：

字　　节	类　　型	描　　述
0	int	文件代码（0x1234，小端）
4	double	x的最小值（小端）
12	double	y的最小值（小端）
20	double	x的最大值（小端）
28	double	y的最大值（小端）
36	int	三角形数量（小端）

紧跟在这个文件头之后的是一系列的三角形记录，每条记录编码为如下的形式：

字　　节	类　　型	描　　述
0	int	记录长度（N字节）
4-N	Points	(X,Y)坐标，以浮点数表示

要写入这个文件，可以使用如下的Python代码：

```
import struct

import itertools

def

write_polys(filename, polys):
    # Determine bounding box

flattened = list(itertools.chain(*polys))

min_x = min(x for

x, y in

flattened)

max_x = max(x for

x, y in

flattened)

min_y = min(y for
```

```
x, y in

flattened)


max_y = max(y for

x, y in

flattened)
with

open(filename, 'wb') as

f:


f.write(struct.pack('<iddddi',


0x1234,


min_x, min_y,


max_x, max_y,


len(polys)))
```

```
for

poly in

polys:

size = len(poly) * struct.calcsize('<dd')

f.write(struct.pack('<i', size+4))

for

pt in

poly:

f.write(struct.pack('<dd', *pt))
# Call it with our polygon data

write_polys('polys.bin', polys)
```

要将结果数据回读的话，可以利用struct.unpack()函数写出相似的代码，只是在编写的时候将所执行的操作反转即可（即，用unpack()取

代之前的pack()）。示例如下：

```
import struct

def

read_polys(filename):

with

open(filename, 'rb') as

f:
        # Read the header



header = f.read(40)

file_code, min_x, min_y, max_x, max_y, num_polys = \

struct.unpack('<iddddi', header)

polys = []
```

```
for
n in
range(num_polys):
pbytes, = struct.unpack('<i', f.read(4))
poly = []
for
m in
range(pbytes // 16):
pt = struct.unpack('<dd', f.read(16))
poly.append(pt)
polys.append(poly)
return
polys
```

尽管这份代码能够工作，但是其中混杂了一些read调用、对结构的解包以及其他一些细节，因此代码比较杂乱。如果用这样的代码去处理一个真正的数据文件，很快就会变的更加混乱。因此，很明显需要寻求其他的解决方案来简化其中的一些步骤，将程序员解放出来，把精力集中在更加重要的问题上。

在本节剩余的部分中，我们将逐步构建出一个用来解释二进制数据的高级解决方案，目的是让程序员提供文件格式的高层规范，而将读取文件以及解包所有数据的细节部分都隐藏起来。先提前给读者预警，本节后面的代码可能是本书中最为高级的示例，运用了多种面向对象编程和元编程的技术。请确保仔细阅读本节的讨论部分，并且需要来回翻阅其他章节，交叉参考。

首先，当我们读取二进制数据时，文件中包含有文件头和其他的数据结构是非常常见的。尽管struct模块能够将数据解包为元组，但另一种表示这种信息的方式是通过类。下面的代码正是这么做的：

```
import struct

class StructField

:
    '''

    Descriptor representing a simple structure field

    '''



def

__init__(self, format, offset):

self.format = format

self.offset = offset

def

__get__(self, instance, cls):

```

```
if

instance is

None:


return

self


else

:



r = struct.unpack_from(self.format,



instance._buffer, self.offset)



return

r[0] if

len(r) == 1 else

r

class Structure
```

```
:

def

__init__(self, bytedata):

self._buffer = memoryview(bytedata)
```

代码中使用了描述符（descriptor）来代表每一个结构字段。每个描述符中都包含了一个struct模块可识别的格式代码（format）以及相对于底层内存缓冲区的字节偏移（offset）。在__get__()方法中，通过struct.unpack_from()函数从缓冲区中解包出对应的值，这样就不用创建额外的切片对象或者执行拷贝动作了。

Structure类只是用作基类，它接受一些字节数据并保存在由StructField描述符所使用的底层内存缓冲区中。这样一来，在Structure类中用到的memoryview()，意图就非常清楚了。

使用这份代码，现在就可以将结构定义为高层次的类，将前面表格中用来描述文件格式的信息都映射到类的定义中。示例如下：

```
class PolyHeader

(Structure):


file_code = StructField('<i', 0)


min_x = StructField('<d', 4)


min_y = StructField('<d', 12)


max_x = StructField('<d', 20)


max_y = StructField('<d', 28)


num_polys = StructField('<i', 36)
```

下面的示例使用这个类来读取之前写入的三角形数据的文件头：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
```

```
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

这么做挺有趣的，但是这种方法还存在许多问题。第一，尽管得到了便利的类接口，但代码比较冗长，需要用户指定许多底层的细节（比如，重复使用StructField、指定偏移量等）。得到的结果中，这个类也缺少一些常用的便捷方法，比如提供一种方式来计算结构的总大小。

任何时候当面对这种过于冗长的类定义时，都应该考虑使用类装饰器（class decorator）或者元类（metaclass）。元类的功能之一是它可用来填充许多底层的实现细节，把这份负担从用户身上拿走。举个例子，考虑下面这个元类和稍微修改过的Structure类：

```
class StructureMeta

(type):
    '''
```

```
    Metaclass that automatically creates StructField descriptors

    '''

def

__init__(self, clsname, bases, clsdict):

fields = getattr(self, '_fields_', [])

byte_order = ''

offset = 0

for

format, fieldname in

fields:

if

format.startswith(('<','>','!','@')):
```

```
byte_order = format[0]

format = format[1:]

format = byte_order + format

setattr(self, fieldname, StructField(format, offset))

offset += struct.calcsize(format)

setattr(self, 'struct_size', offset)
class Structure

(metaclass=StructureMeta):

def

__init__(self, bytedata):

self._buffer = bytedata

@classmethod
```

```
def
from_file(cls, f):
return
cls(f.read(cls.struct_size))
```

使用这个新的Structure类，现在就可以像这样编写结构的定义了：

```
class PolyHeader
(Structure):
_fields_ = [
   ('<i', 'file_code'),
   ('d', 'min_x'),
('d', 'min_y'),
```

```
('d', 'max_x'),

('d', 'max_y'),

('i', 'num_polys')

]
```

可以看到，现在的定义要简化得多。新增的类方法from_file()也使得从文件中读取数据变得更加简单，因为现在不需要了解数据的结构大小等细节问题了。比如，现在可以这么做：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

一旦引入了元类，就可以为其构建更多智能化的操作。比如说，假设想对嵌套型的二进制结构提供支持。下面是对这个元类的修改，以及对新功能提供支持的描述符定义：

```
class NestedStruct

:
    '''

    Descriptor representing a nested structure

    '''




def

__init__(self, name, struct_type, offset):

self.name = name

self.struct_type = struct_type

```

```
self.offset = offset

def

__get__(self, instance, cls):

if

instance is

None:

return

self

else

:

data = instance._buffer[self.offset:

self.offset+self.struct_type.struct_size]
```

```
result = self.struct_type(data)
            # Save resulting structure back on instance to avoid

            # further recomputation of this step

setattr(instance, self.name, result)

return

result
class StructureMeta

(type):
    '''

    Metaclass that automatically creates StructField descriptors

    '''

def

__init__(self, clsname, bases, clsdict):
```

```
fields = getattr(self, '_fields_', [])

byte_order = ''

offset = 0

for

format, fieldname in

fields:

if

isinstance(format, StructureMeta):

setattr(self, fieldname,

NestedStruct(fieldname, format, offset))

offset += format.struct_size

else
```

```
:

if

format.startswith(('<','>','!','@')):

byte_order = format[0]

format = format[1:]

format = byte_order + format

setattr(self, fieldname, StructField(format, offset))

offset += struct.calcsize(format)

setattr(self, 'struct_size', offset)
```

在这份代码中，NestedStruct描述符的作用是在一段内存区域上定义另一个结构 [2] 。这是通过在原内存缓冲区中取一个切片，然后在这个切片上实例化给定的结构类型来实现的。由于底层的内存缓

冲区是由memoryview来初始化的，因此这个切片操作不会涉及任何额外的内存拷贝动作。相反，它只是在原来的内存中“覆盖”上新的结构实例。此外，要避免重复的实例化动作，这个描述符会利用8.10节中提到的技术将内层结构对象保存在该实例上。

使用这种新的技术，现在就可以像这样编写代码了：

```
class Point

(Structure):


_fields_ = [


('<d', 'x'),


('d', 'y')


]
class PolyHeader

(Structure):


_fields_ = [
```

```
  ('<i', 'file_code'),

  (Point, 'min'), # nested struct

  (Point, 'max'), # nested struct

 ('i', 'num_polys')

]
```

太神奇了，一切都还是按照所期望的方式正常运转。示例如下：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min # Nested structure

<__main__.Point object at 0x1006a48d0>
>>> phead.min.x
```

```
0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
>>>
```

到目前为止，我们已经成功开发了一个用来处理固定大小记录的框架。但是对于大小可变的组件又该如何处理呢？比如说，这份三角形数据文件的剩余部分中包含有大小可变的区域。

一种处理方法是编写一个类来简单代表一块二进制数据，并附带一个通用函数来负责以不同的方式来解释数据的内容。这和6.11节中的代码关系紧密：

```
class SizedRecord

:

def

__init__(self, bytedata):
```

```
self._buffer = memoryview(bytedata)

@classmethod

def

from_file(cls, f, size_fmt, includes_size=True):

sz_nbytes = struct.calcsize(size_fmt)

sz_bytes = f.read(sz_nbytes)

sz, = struct.unpack(size_fmt, sz_bytes)

buf = f.read(sz - includes_size * sz_nbytes)

return

cls(buf)


def

iter_as(self, code):
```

```
if
isinstance(code, str):
s = struct.Struct(code)
for
off in
range(0, len(self._buffer), s.size):
yield
s.unpack_from(self._buffer, off)
elif
isinstance(code, StructureMeta):
size = code.struct_size
for
```

```
off in

range(0, len(self._buffer), size):

data = self._buffer[off:off+size]

yield

code(data)
```

这里的类方法SizedRecord.from_file()是一个通用的函数，用来从文件中读取大小预定好的数据块，这在许多文件格式中都是很常见的。对于输入参数，它可接受结构的格式代码，其中包含有编码的大小（以字节数表示）。可选参数includes_size用来指定字节数中是否要包含进文件头的大小。下面的示例展示如何使用这份代码来读取三角形数据文件中那些单独的三角形：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...                 for

n in
```

```
range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
 <__main__.SizedRecord object at 0x1006a4f50>,
 <__main__.SizedRecord object at 0x10070da90>]
>>>
```

可以看到，SizedRecord实例的内容还没有经过解释。要做到这一点，可以使用iter_as()方法。该方法可接受一个结构格式代码或者Structure类作为输入。这给了我们极大的自由来选择如何解释数据。比如：

```
>>> for

n, poly in

enumerate(polydata):
...     print

('Polygon', n)
...     for

p in

poly.iter_as('<dd'):
...         print

(p)
```

```
...

Polygon 0
(1.0, 2.5)
(3.5, 4.0)
(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

>>> for

n, poly in

enumerate(polydata):
...     print

('Polygon', n)
...     for

p in

poly.iter_as(Point):
...             print

(p.x, p.y)
...
```

```
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2
3.4 6.3
1.2 0.5
4.6 9.2
>>>
```

现在我们把所有的东西结合起来。下面是read_polys()函数的另一种实现：

```
class Point


(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]
class PolyHeader


(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]
```

```
def

read_polys(filename):
    polys = []
    with

open(filename, 'rb') as

f:
        phead = PolyHeader.from_file(f)
        for

n in

range(phead.num_polys):
            rec = SizedRecord.from_file(f, '<i')
            poly = [ (p.x, p.y)
                      for

p in

rec.iter_as(Point) ]
            polys.append(poly)
    return

polys
```

6.12.3 讨论

本节展示了多种高级编程技术的实际应用，这

些技术包括描述符、惰性求值、元类、类变量以及memoryview。只是它们都用于一个非常具体的目的而已。

本节给出的实现中，一个非常重要的特性就是强烈基于惰性展开（lazy-unpacking）的思想。每当创建出一个Structure实例时，__init__()方法只是根据提供的字节数据创建出一个memoryview，除此之外别的什么都不做。具体而言就是这个时候不会进行任何的解包或其他与结构相关的操作。采用这种方法的一个动机是我们可能只对二进制记录中的某几个特定部分感兴趣。与其将整个文件解包展开，不如只对实际要访问到的那几个部分解包即可。

要实现惰性展开和对值进行打包，StructField描述符就派上用场了。用户在_fields_中列出的每个属性都会转换为一个StructField描述符，它保存着相关属性的结构化代码和相对于底层内存缓冲区的字节偏移量。当我们定义各种各样的结构化类型时，元类StructureMeta用来自动创建出这些描述符。使用元类的主要原因在于这么做能以高层次的描述来指定结构的格式，完全不用操心底层的细节问题，因而能极大地简化用户的操作。

元类StructureMeta中有一个微妙的方面需要注意，那就是它将字节序给规定死了。也就说，如果

有任何属性指定了字节序（<指代小端序，而>指代大端序），那么这个字节序就适用于该属性之后的所有字段。这种行为可避免我们产生额外的键盘输入，同时也使得在定义字段时可以切换字节序。比如说，我们可能会碰到下面这样更加复杂的数据：

```
class ShapeFile


(Structure):
    _fields_ = [ ('>i', 'file_code'), # Big endian 这里是大端，后两



                 ('20s', 'unused'),
                 ('i', 'file_length'),
                 ('<i', 'version'), # Little endian 切换为小端，后约



                 ('i', 'shape_type'),
                 ('d', 'min_x'),
                 ('d', 'min_y'),
                 ('d', 'max_x'),
                 ('d', 'max_y'),
                 ('d', 'min_z'),
                 ('d', 'max_z'),
                 ('d', 'min_m'),
                 ('d', 'max_m') ]
```

前文中提到，解决方案中对memoryview()的使用起到了避免内存拷贝的作用。当结构数据开始出现嵌套时，memoryview可用来在相同的内存区域中

覆盖上不同的结构定义。这种行为十分微妙，它考虑到了切片操作在memoryview和普通的字节数组上的不同行为。如果对字节串或字节数组执行切片操作的话，通常都会得到一份数据的拷贝，但memoryview就不会这样——切片只是简单地覆盖在已有的内存之上。因此，这种方法更加高效。

还有一些相关的章节会帮助我们对解决方案中用到的技术进行扩展。8.13节中采用描述符构建了一个类型系统。8.10节中介绍了有关惰性计算的性质，这个和NestedStruct描述符的实现有一定的相关性。9.19节中有一个例子采用元类来初始化类的成员，这个和StructureMeta类采用的方式非常相似。我们可能也会对Python标准库中ctypes模块的源代码产生兴趣，因为它对定义数据结构、对数据结构的嵌套以及类似功能的支持和我们的解决方案比较相似。

6.13 数据汇总和统计

6.13.1 问题

我们需要在大型数据库中查询数据并由此生成汇总或者其他形式的统计数据。

6.13.2 解决方案

对于任何涉及统计、时间序列以及其他相关技术的数据分析问题，都应该使用Pandas库（http://pandas.pydata.org ）。

为了小试牛刀，下面这个例子使用Pandas来分析芝加哥的老鼠和啮齿动物数据库（https://data.cityofchicago.org/Service-Requests/311-Service-Requests-Rodent-Baiting/97t6-zrhs ）。在写作本书时，这个CSV文件中有大约74 000条数据：

```
>>> import pandas

>>> # Read a CSV file, skipping last line

>>> rats = pandas.read_csv('rats.csv', skip_footer=1)
```

```
>>> rats
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74055 entries, 0 to 74054
Data columns:
Creation Date                74055 non-null values
Status                    74055 non-null values
Completion Date                72154 non-null values
Service Request Number         74055 non-null values
Type of Service Request         74055 non-null values
Number of Premises Baited         65804 non-null values
Number of Premises with Garbage    65600 non-null values
Number of Premises with Rats     65752 non-null values
Current Activity             66041 non-null values
Most Recent Action             66023 non-null values
Street Address                74055 non-null values
ZIP Code                     73584 non-null values
X Coordinate                 74043 non-null values
Y Coordinate                 74043 non-null values
Ward                         74044 non-null values
Police District                74044 non-null values
Community Area                74044 non-null values
Latitude                     74043 non-null values
Longitude                     74043 non-null values
Location                     74043 non-null values
dtypes: float64(11), object(9)
>>> # Investigate range of values for a certain field



>>> rats['Current Activity'].unique()
array([nan, Dispatch Crew, Request Sanitation Inspector], dtype=
>>> # Filter the data



>>> crew_dispatched = rats[rats['Current Activity'] == 'Dispatch
>>> len(crew_dispatched)
65676
>>>
>>> # Find 10 most rat-infested ZIP codes in Chicago



>>> crew_dispatched['ZIP Code'].value_counts()[:10]
60647          3837
```

```
60618         3530
60614         3284
60629         3251
60636         2801
60657         2465
60641         2238
60609         2206
60651         2152
60632         2071
>>>
>>> # Group by completion date



>>> dates = crew_dispatched.groupby('Completion Date')
<pandas.core.groupby.DataFrameGroupBy object at 0x10d0a2a10>
>>> len(dates)
472
>>>
>>> # Determine counts on each day



>>> date_counts = dates.size()
>>> date_counts[0:10]
Completion Date
01/03/2011            4
01/03/2012          125
01/04/2011           54
01/04/2012           38
01/05/2011           78
01/05/2012          100
01/06/2011          100
01/06/2012           58
01/07/2011            1
01/09/2012           12
>>>

>>> # Sort the counts



>>> date_counts.sort()
>>> date_counts[-10:]
Completion Date
```

```
10/12/2012              313
10/21/2011              314
09/20/2011              316
10/26/2011              319
02/22/2011              325
10/26/2012              333
03/17/2011              336
10/13/2011              378
10/14/2011              391
10/07/2011              457
>>>
```

你没看错，2011年10月7号对于老鼠来说的确是非常忙碌的一天。

6.13.3　讨论

Pandas是一个庞大的库，它还有更多的功能，但我们无法在此一一描述。但是，如果需要分析大型的数据集、将数据归组、执行统计分析或者其他类似的任务，那么Pandas绝对值得一试。

由Wes McKinney所著的Python for Data Analysis（O’Reilly）一书中也包含了更多的内容。

---

[1]　Num-Premises中的-不能用作Python的标识符字符。——译者注

[2] 类似C++中的placement new技法。——译者注

第7章　函数

用def语句定义的函数是所有程序的基石。本章的目的是向读者展示一些更加高级和独特的函数定义以及使用模式。主题包括默认参数、可接受任意数量参数的函数、关键字参数、参数注解以及闭包。此外，有关利用回调函数实现巧妙的控制流以及数据传递的问题也有涉及。

7.1　编写可接受任意数量参数的函数

7.1.1　问题

我们想编写一个可接受任意数量参数的函数。

7.1.2　解决方案

要编写一个可接受任意数量的位置参数的函数，可以使用以*开头的参数。示例如下：

```
def

avg(first, *rest):
return

(first + sum(rest)) / (1 + len(rest))

# Sample use

avg(1, 2)             # 1.5

avg(1, 2, 3, 4)     # 2.5
```

在这个示例中，rest是一个元组，它包含了其他所有传递过来的位置参数。代码在之后的计算中会将其视为一个序列来处理。

如果要接受任意数量的关键字参数，可以使用以**开头的参数。示例如下：

```
import html

def

make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for

item in

attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
                  name=name,
                  attrs=attr_str,
                  value=html.escape(value))
    return

element
```

```
# Example

# Creates '<item size="large" quantity="6">Albatross</item>'

make_element('item', 'Albatross', size='large', quantity=6)
# Creates '<p>&lt;spam&gt;</p>'

make_element('p', '<spam>')
```

在这里attrs是一个字典，它包含了所有传递过来的关键字参数（如果有的话）。

如果想要函数能同时接受任意数量的位置参数和关键字参数，只要联合使用*和**即可。示例如下：

```
def

anyargs(*args, **kwargs):
    print

(args)       # A tuple

    print
```

```
(kwargs)    # A dict
```

在这个函数中，所有的位置参数都会放置在元组args中，而所有的关键字参数都会放置在字典kwargs中。

7.1.3 讨论

在函数定义中，以*打头的参数只能作为最后一个位置参数出现，而以**打头的参数只能作为最后一个参数出现。在函数定义中存在一个很微妙的特性，那就是在*打头的参数后仍然可以有其他的参数出现。

```
def

a(x, *args, y):
    pass

def

b(x, *args, y, **kwargs):
    pass
```

这样的参数称之为keyword-only参数（即，出现在*args之后的参数只能作为关键字参数使用）。7.2节中会做进一步的讨论。

7.2　编写只接受关键字参数的函数

7.2.1　问题

我们希望函数只通过关键字的形式接受特定的参数。

7.2.2　解决方案

如果将关键字参数放置在以*打头的参数或者是一个单独的*之后，这个特性就很容易实现。示例如下：

```
def

recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True)        # TypeError

recv(1024, block=True)    # Ok
```

这项技术也可以用来为那些可接受任意数量的位置参数的函数来指定关键字参数。示例如下:

```
def

mininum(*values, clip=None):
    m = min(values)
    if

clip is not

None:
        m = clip if

clip > m else

m
    return

m

minimum(1, 5, 2, -5, 10)            # Returns -5



minimum(1, 5, 2, -5, 10, clip=0)     # Returns 0
```

7.2.3 讨论

当指定可选的函数参数时，keyword-only参数常常是一种提高代码可读性的好方法。比如，考虑下面这个调用：

```
msg = recv(1024, False)
```

如果某些人不熟悉recv()的工作方式，他们可能会搞不清楚这里的False参数到底表示了什么意义。而另一方面，如果这个调用可以写成下面这样的话，那就显得清晰多了：

```
msg = recv(1024, block=False)
```

在有关**kwargs的技巧中，使用keyword-only参数常常也是很可取的。因为当用户请求帮助信息时，它们可以适时地显示出来：

```
>>> help(recv)
Help on function recv in module __main__:
recv(maxsize, *, block)
    Receives a message
```

keyword-only参数在更加高级的上下文环境中

同样也能起到作用。比如说，可以用来为函数注入参数，这些函数利用*args和**kwargs接受所有的输入参数。可参见9.11节中的示例。

7.3 将元数据信息附加到函数参数上

7.3.1 问题

我们已经编写好了一个函数，但是希望能为参数附加上一些额外的信息，这样其他人可以对函数的使用方法有更多的认识和了解。

7.3.2 解决方案

函数的参数注解可以提示程序员该函数应该如何使用，这是很有帮助的。比如说，考虑下面这个带参数注解的函数：

```
def add(x:int, y:int) -> int:
    return x + y
```

Python解释器并不会附加任何语法意义到这些

参数注解上。它们既不是类型检查也不会改变Python的行为。但是，参数注解会给其他阅读源代码的人带来有用的提示。一些第三方工具和框架可能也会为注解加上语法含义。这些注解也会出现在文档中：

```
>>> help(add)
Help on function add in module __main__:

add(x: int, y: int) -> int
>>>
```

尽管可以将任何类型的对象作为函数注解附加到函数定义上（比如，数字、字符串、实例等），但是通常只有类和字符串才显得最有意义。

7.3.3 讨论

函数注解只会保存在函数的__annotations__属性中。示例如下：

```
>>> add.__annotations__
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

尽管函数注解有着许多潜在的用途，但它们的主要功能也许就是丰富一下文档内容了。因为

Python中并没有类型声明，所以如果只是简单地阅读一下源代码就想知道打算给函数传递什么对象常常是比较困难的。函数注解就可以带给我们更多的提示。

请参见9.20节中的高级示例，那个例子展示了如何利用函数注解来实现函数重载。

7.4 从函数中返回多个值

7.4.1 问题

我们想从函数中返回多个值。

7.4.2 解决方案

要从函数中返回多个值，只要简单地返回一个元组即可。示例如下：

```
>>> def


myfun():
...     return


1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

7.4.3 讨论

尽管看起来myFun()返回了多个值，但实际上它只创建了一个元组而已。这看起来有点奇怪，但是实际上元组是通过逗号来组成的，不是那些圆括号。示例如下：

```
>>> a = (1, 2)         # With parentheses
>>> a
(1, 2)
>>> b = 1, 2           # Without parentheses
>>> b
(1, 2)
>>>
```

当调用的函数返回了元组，通常会将结果赋值给多个变量，就像示例中那样。实际上这就是简单的元组解包，我们在1.1节中就已经提到过了。返回的值也可以只赋给一个单独的变量：

```
>>> x = myfun()
>>> x
(1, 2, 3)
>>>
```

这样x就代表整个元组。

7.5　定义带有默认参数的函数

7.5.1　问题

我们想定义一个函数或者方法，其中有一个或多个参数是可选的并且带有默认值。

7.5.2　解决方案

表面上看定义一个带有可选参数的函数是非常简单的——只需要在定义中为参数赋值，并确保默认参数出现在最后即可。示例如下：

```
def

spam(a, b=42):
    print

(a, b)

spam(1)          # Ok. a=1, b=42

spam(1, 2)          # Ok. a=1, b=2
```

如果默认值是可变容器的话，比如说列表、集合或者字典，那么应该把None作为默认值，代码应该像这样编写：

```
# Using a list as a default value

def

spam(a, b=None):
    if

b is

None:
        b = []
...
```

如果不打算提供一个默认值，只是想编写代码来检测可选参数是否被赋予了某个特定的值，那么可以采用下面的惯用手法：

```
_no_value = object()

def

spam(a, b=_no_value):
```

```
    if

b is

_no_value:
        print

('No b value supplied')
    ...
```

这个函数的行为是这样的：

```
>>> spam(1)
No b value supplied
>>> spam(1, 2)        # b = 2



>>> spam(1, None)  # b = None



>>>
```

请仔细区分不传递任何值和传递None之间的区别。

7.5.3　讨论

定义带有默认参数的函数看似很容易，但其实

并不像看到的那么简单。

首先，对默认参数的赋值只会在函数定义的时候绑定一次。可用下面这个例子做下试验：

```
>>> x = 42
>>> def


spam(a, b=x):
...     print


(a, b)
...




>>> spam(1)
1 42
>>> x = 23    # Has no effect



>>> spam(1)
1 42
>>>
```

注意到修改变量x的值（x被作为函数参数的默认值）并没有对函数产生任何效果。这是因为默认值已经在函数定义的时候就确定好了。

其次，给默认参数赋值的应该总是不可变的对象，比如None、True、False、数字或者字符串。特

别要注意的是，绝对不要编写这样的代码：

```
def

spam(a, b=[]):        # NO!


    ...
```

如果这么做了就会陷入到各种麻烦之中。如果默认值在函数体之外被修改了，那么这种修改将在之后的函数调用中对参数的默认值产生持续的影响。示例如下：

```
>>> def

spam(a, b=[]):
...

     print

(b)
...

     return

b
...

```

```
>>> x = spam(1)
>>> x
[]
>>> x.append(99)
>>> x.append('Yow!')
>>> x
[99, 'Yow!']
>>> spam(1)             # Modified list gets returned!



[99, 'Yow!']
>>>
```

这很可能不是所期望的结果。要避免出现这种问题，最好按照解决方案中的做法，使用None作为默认值并在函数体中增加一个对默认值的检查。

当检测默认参数是否为None时，本节示例的关键之处在于对is操作符的运用。有时候人们会犯这样的错误：

```
def


spam(a, b=None):
    if not


b:       # NO! Use 'b is None' instead
```

```
        b = []
...
```

这里出现的问题在于尽管None会被判定为False，可是还有许多其他的对象（比如长度为0的字符串、列表、元组、字典等）也存在这种行为。因此，上面示例给出的条件检测会将某些特定的输入也判定为False，从而错误地忽略掉这些输入值。示例如下：

```
>>> spam(1)         # OK



>>> x = []
>>> spam(1, x)         # Silent error. x value overwritten by de



>>> spam(1, 0)         # Silent error. 0 ignored



>>> spam(1, '')         # Silent error. '' ignored



>>>
```

本节最后讨论的内容更加巧妙——在函数中检测是否对可选参数提供了某个特定值（可以是任意值）。这里最为棘手的地方在于我们不能用None、

0或者False当做默认值来检测用户是否提供了参数（因为所有这些值都是完全合法的参数，用户极有可能将它们当做参数）。因此，需要用其他的办法来检测。

要解决这个问题，可以利用object()创建一个独特的私有实例，就像解决方案中给出的那样（即，变量_no_value）。在函数中，可以用这个特殊值来同用户提供的参数做相等性检测，以此判断用户是否提供了参数。这里主要考虑到对于用户来说，把_no_value实例作为输入参数几乎是不可能的。因此，如果要判断用户是否提供了某个参数，_no_value就成了一个可以用来安全比较的值。

这里用到的object()可能看起来很不常见。object作为Python中几乎所有对象的基类而存在。可以创建object的实例，但是它们没有任何值得注意的方法，也没有任何实例数据，因此一般来说我们对它是毫无兴趣的（因为底层缺少__dict__字典，我们甚至没法为它设置任何属性）。唯一可做的就是检测相等性，这也使得它们可作为特殊值来使用，就像我们给出的解决方案中那样。

7.6　定义匿名或内联函数

7.6.1　问题

我们需要提供一个短小的回调函数为sort()这样的操作所用，但是又不想通过def语句编写一个单行的函数。相反，我们更希望能有一种简便的方式来定义“内联”式的函数。

7.6.2　解决方案

像这种仅仅完成表达式求值的简单函数可以通过lambda表达式来替代。示例如下：

```
>>> add = lambda


x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

这里用到的lambda表达式与下面的函数定义有着相同的功能：

```
>>> def


add(x, y):
...


     return


x + y
...



>>> add(2,3)
5
>>>
```

一般来说，lambda表达式可用在如下的上下文环境中，比如排序或者对数据进行整理时：

```
>>> names = ['David Beazley', 'Brian Jones',
...


           'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda


name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian J
>>>
```

7.6.3 讨论

尽管lambda表达式允许定义简单的函数，但它的局限性也很大。具体来说，我们只能指定一条单独的表达式，这个表达式的结果就是函数的返回值。这意味着其他的语言特性比如多行语句、条件分支、迭代和异常处理统统都无法使用。

不使用lambda表达式也可以愉快地编写出大量的Python代码。但是，还是时不时会在一些程序中见到lambda的身影。比如有的人会编写很多微型函数来对各种表达式进行求值，或者在需要用户提供回调函数的时候，这时lambda表达式就能派上用场了。

7.7　在匿名函数中绑定变量的值

7.7.1　问题

我们利用lambda表达式定义了一个匿名函数，但是也希望可以在函数定义的时候完成对特定变量的绑定。

7.7.2　解决方案

考虑下列代码的行为：

```
>>> x = 10
>>> a = lambda


y: x + y
>>> x = 20
>>> b = lambda


y: x + y
>>>
```

现在请问自己一个问题，a(10)和b(10)的结果是多少？如果觉得结果是20和30的话，那就错了：

```
>>> a(10)
```

```
30
>>> b(10)
30
>>>
```

这里的问题在于lambda表达式中用到的x是一个自由变量，在运行时才进行绑定而不是定义的时候绑定。因此，lambda表达式中x的值应该是在执行时确定的，执行时x的值是多少就是多少。示例如下：

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

如果希望匿名函数可以在定义的时候绑定变量，并保持值不变，那么可以将那个值作为默认参数实现，就像下面这样：

```
>>> x = 10
>>> a = lambda

y, x=x: x + y
>>> x = 20
>>> b = lambda
```

```
y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

7.7.3 讨论

本节中提到的问题一般比较容易出现在那些对lambda函数过于“聪明”的应用上。比方说，通过列表推导来创建一列lambda表达式，或者在一个循环中期望lambda表达式能够在定义的时候记住迭代变量。示例如下：

```
>>> funcs = [lambda

x: x+n for

n in

range(5)]
>>> for

f in

funcs:
...

    print
```

```
(f(0))
...



4
4
4
4
4
>>>
```

我们可以注意到所有的函数都认为n的值为4，也就是迭代中的最后一个值。我们再和下面的代码做下对比：

```
>>> funcs = [lambda

x, n=n: x+n for

n in

range(5)]
>>> for

f in

funcs:
...

    print
```

```
(f(0))
...



0
1
2
3
4
>>>
```

可以看到，现在函数可以在定义的时候捕获到n的值了。

7.8 让带有N个参数的可调用对象以较少的参数形式调用

7.8.1 问题

我们有一个可调用对象可能会以回调函数的形式同其他的Python代码交互。但是这个可调用对象需要的参数过多，如果直接调用的话会产生异常。

7.8.2 解决方案

如果需要减少函数的参数数量，应该使用functools.partial()。函数partial()允许我们给一个或多个参数指定固定的值，以此减少需要提供给之后调用的参数数量。为了说明这个过程，假设有这么一个函数：

```
def

spam(a, b, c, d):
    print

(a, b, c, d)
```

现在考虑用partial()来对参数赋固定的值：

```
>>> from functools import


partial
>>> s1 = partial(spam, 1)         # a = 1



>>> s1(2, 3, 4)
1 2 3 4
>>> s1(4, 5, 6)
1 4 5 6
>>> s2 = partial(spam, d=42)     # d = 42



>>> s2(1, 2, 3)
1 2 3 42
>>> s2(4, 5, 5)
4 5 5 42
>>> s3 = partial(spam, 1, 2, d=42)     # a = 1, b = 2, d = 42



>>> s3(3)
1 2 3 42
>>> s3(4)
1 2 4 42
>>> s3(5)
1 2 5 42
>>>
```

我们可以观察到partial()对特定的参数赋了固定值并返回了一个全新的可调用对象。这个新的可调用对象仍然需要通过指定那些未被赋值的参数来调用。这个新的可调用对象将传递给partial()的固定参

数结合起来，统一将所有的参数传递给原始的函数。

7.8.3 讨论

本节提到的技术对于将看似不兼容的代码结合起来使用是大有裨益的。下面我们用一系列的示例来帮助理解。

第一个例子是，假设有一列以元组(x, y)来表示的点坐标。可以用下面的函数来计算两点之间的距离：

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

import math



def


distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return


math.hypot(x2 - x1, y2 - y1)
```

现在假设想根据这些点之间的距离来对它们排

序。列表的sort()方法可接受一个key参数，它可用来做自定义的排序处理。但是它只能和接受单参数的函数一起工作（因此和distance()是不兼容的）。下面我们用partical()来解决这个问题：

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

我们可以对这个思路进行扩展，partial()常常可用来调整其他库中用到的回调函数的参数签名。比方说，这里有一段代码利用multiprocessing模块以异步方式计算某个结果，并将这个结果传递给一个回调函数。该回调函数可接受这个结果以及一个可选的日志参数：

```
def

output_result(result, log=None):
    if

log is not

None:
        log.debug('Got: %r', result)

# A sample function
```

```
def

add(x, y):
    return

x + y

if

__name__ == '__main__':
    import logging

    from multiprocessing import

Pool
    from functools import

partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')

    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, lo
    p.close()
    p.join()
```

当我们在apply_async()中指定回调函数时，额外的日志参数是通过partical()来指定的。multiprocessing 模块对于这些细节根本一无所知

——它只通过单个参数来调用回调函数。

作为类似的例子，考虑一下我们在编写网络服务器程序时面对的问题。有了socketserver模块，这一切相对来说都变得很简单了。比方说，下面有一个简单的echo服务程序：

```
from socketserver import


StreamRequestHandler, TCPServer

class EchoHandler


(StreamRequestHandler):
    def


handle(self):
        for


line in


self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

现在，假设我们想在EchoHandler类中增加一个__init__()方法，让它接受一个额外的配置参数。

示例如下：

```
class EchoHandler


(StreamRequestHandler):
    # ack is added keyword-only argument. *args, **kwargs are



    # any normal parameters supplied (which are passed on)



    def


__init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)
    def


handle(self):
        for


line in


self.rfile:
            self.wfile.write(self.ack + line)
```

如果做了上述改动，现在就会发现没法简单地将其插入到TCPServer类中了。事实上，你会发现代码会产生如下的异常：

```
Exception happened during processing of request from ('127.0.0.1
Traceback (most recent call last):
...
TypeError: __init__() missing 1 required keyword-only argument:
```

初看上去，除了修改socketserver的源代码或者采用一些拐弯抹角的技法外，似乎没别的办法修正这份代码了。但是，利用partical()就能轻松解决这个问题。只用在partial()中提供ack的参数值即可，就像下面这样：

```
from functools import

partial
serv = TCPServer(('', 15000), partial(EchoHandler, ack=b'RECEIVEI
serv.serve_forever()
```

在这个例子里，__init__()方法中对参数ack的指定看起来有些滑稽，但它是以keyword- only参数的形式来指定的。有关keyword-only参数的讨论可以在7.2节中找到。

有时候也可以通过lambda表达式来替代partial()。比如，上面这几个例子也可以采用这样的语句来实现：

```
points.sort(key=lambda


p: distance(pt, p))

p.apply_async(add, (3, 4), callback=lambda


result: output_result(result,log))

serv = TCPServer(('', 15000),
                 lambda


*args, **kwargs: EchoHandler(*args,
                                                     ack=b'RECEIV
                                                     **kwargs))
```

这些代码也能正常工作，但是却显得很啰嗦，而且也让人觉得读起来很困惑。使用partial()会使得你的意图更加明确（即，为某些参数提供默认值）。

7.9 用函数替代只有单个方法的类

7.9.1 问题

我们有一个只定义了一个方法的类（除__init__()方法外）。但是，为了简化代码，我们更希望能够只用一个简单的函数来替代。

7.9.2 解决方案

在许多情况下，只有单个方法的类可以通过闭包（closure）将其转换成函数。考虑下面这个例子，这个类允许用户通过某种模板方案来获取URL。

```
from urllib.request import


urlopen

class UrlTemplate


:
    def


__init__(self, template):
        self.template = template
    def
```

```
open(self, **kwargs):
        return

urlopen(self.template.format_map(kwargs))

# Example use. Download stock data from yahoo

yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={nar
for

line in

yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):
    print

(line.decode('utf-8'))
```

这个类可以用一个简单的函数来取代：

```
def

urltemplate(template):
    def

opener(**kwargs):
        return

urlopen(template.format_map(kwargs))
    return
```

```
opener

# Example use



yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={nar
for


line in


yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):
    print


(line.decode('utf-8'))
```

7.9.3 讨论

在许多情况下，我们会使用只有单个方法的类的唯一原因就是保存额外的状态给类方法使用。比方说，UrlTemplate类的唯一目的就是将template的值保存在某处，这样就可以在open()方法中用上它了。

按照我们给出的解决方案，使用嵌套函数或者说闭包常常会显得更加优雅。简单来说，闭包就是一个函数，但是它还保存着额外的变量环境，使得这些变量可以在函数中使用。闭包的核心特性就是

它可以记住定义闭包时的环境。因此，在这个解决方案中，opener()函数可以记住参数template的值，然后在随后的调用中使用该值。

无论何时，当在编写代码中遇到需要附加额外的状态给函数时，请考虑使用闭包。比起将函数放入一个“全副武装”的类中，基于闭包的解决方案通常更加简短也更加优雅。

7.10　在回调函数中携带额外的状态

7.10.1　问题

我们正在编写需要使用回调函数的代码（比如，事件处理例程、完成回调等），但是希望回调函数可以携带额外的状态以便在回调函数内部使用。

7.10.2　解决方案

本节中提到的对回调函数的应用可以在许多库和框架中找到——尤其是那些和异步处理相关的库和框架。为了说明和测试的目的，我们首先定义下面的函数，它会调用一个回调函数：

```
def

apply_async(func, args, *, callback):
    # Compute the result

    result = func(*args)

    # Invoke the callback with the result
```

```
    callback(result)
```

在现实世界中，类似这样的代码可能会完成各种高级的处理任务，这会涉及线程、进程和定时器等，但我们这里主要关注的不是这些。相反，我们只是把注意力集中在对回调函数的调用上。下面的示例展示了上述代码应该如何使用：

```
>>> def


print_result(result):
...


     print


('Got:', result)
...



>>> def


add(x, y):
...


     return


x + y
...
```

```
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Got: helloworld
>>>
```

我们会注意到函数print_result()仅接受一个单独的参数，也就是result。这里并没有传入其他的信息到函数中。有时候当我们希望回调函数可以同其他变量或者部分环境进行交互时，缺乏这类信息就会带来问题。

一种在回调函数中携带额外信息的方法是使用绑定方法（bound-method）而不是普通的函数。比如，下面这个类保存了一个内部的序列号码，每当接收到一个结果时就递增这个号码。

```
class ResultHandler


:
    def


__init__(self):
        self.sequence = 0
    def


handler(self, result):
```

```
        self.sequence += 1
        print


('[{}] Got: {}'.format(self.sequence, result))
```

要使用这个类，可以创建一个类实例并将绑定方法handler当做回调函数来用：

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>
```

作为类的替代方案，也可以使用闭包来捕获状态。示例如下：

```
def


make_handler():
    sequence = 0
    def


handler(result):
        nonlocal sequence
        sequence += 1
        print


('[{}] Got: {}'.format(sequence, result))
```

```
return

handler
```

下面是使用闭包的例子：

```
>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>
```

除此之外还有一种方法，有时候可以利用协程（coroutine）来完成同样的任务：

```
def

make_handler():
    sequence = 0
    while

True:
        result = yield

        sequence += 1
        print

('[{}] Got: {}'.format(sequence, result))
```

```
```

对于协程来说，可以使用它的send()方法来作为回调函数，就像下面这样：

```
>>> handler = make_handler()
>>> next(handler)          # Advance to the yield



>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

最后但也同样重要的是，也可以通过额外的参数在回调函数中携带状态，然后用partial()来处理参数个数的问题（见7.8节）。示例如下：

```
>>> class SequenceNo


:
...     def


__init__(self):
...         self.sequence = 0
...
>>> def


handler(result, seq):
...     seq.sequence += 1
```

```
...     print


('[{}] Got: {}'.format(seq.sequence, result))
...
>>> seq = SequenceNo()
>>> from functools import


partial
>>> apply_async(add, (2, 3), callback=partial(handler, seq=seq))
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=partial(handle
[2] Got: helloworld
>>>
```

7.10.3 讨论

基于回调函数的软件设计常常会面临使代码陷入一团乱麻的风险。部分原因是因为从代码发起初始请求开始到回调执行的这个过程中，回调函数常常是与这个环境相脱离的。因此，在发起请求和处理结果之间的执行环境就丢失了。如果想让回调函数在涉及多个步骤的任务处理中能够继续执行，就必须清楚应该如何保存和还原相关的状态。

主要有两种方法可用于捕获和携带状态。可以在类实例上携带状态（将状态附加到绑定方法上），也可以在闭包中携带状态。这两种方法中，闭包可能要显得更轻量级一些，而且由于闭包也是由函数构建的，这样显得会更加自然。这两种方法

都可以自动捕获所有正在使用的变量。因此，这就使得我们不必担心哪个具体的状态需要保存起来（根据代码自动决定哪些需要保存）。

如果使用闭包，那么需要对可变变量多加留意。在给出的解决方案中，nonlocal声明用来表示变量sequence是在回调函数中修改的。没有这个声明，将得到错误提示。

将协程用作回调函数的有趣之处在于这种方式和采用闭包的方案关系紧密。从某种意义上说，协程甚至更为清晰，因为这里只出现了一个单独的函数。此外，变量都可以自由地进行修改，不必担心nonlocal声明。可能存在的缺点在于人们对协程的理解程度不如其他的Python特性。使用协程时还有几个小技巧需要掌握，比如在使用协程前需要先对其调用一次next()，这在实践中常常容易忘记。不过，协程还有其他的潜在用途，比如定义内联的回调函数（在下一节中讲解）。

如果所有需要做的就是在回调函数中传入额外的值，那么最后提到的那个有关partial()的技术是很管用的。有时候我们也会看到用lambda表达式来实现同样的功能：

```
>>> apply_async(add, (2, 3), callback=lambda
```

```
r: handler(r, seq))
[1] Got: 5
>>>
```

要查看更多的示例请参见7.8节。在那一节中我们展示了如何利用partial()来修改函数的参数签名。

7.11 内联回调函数

7.11.1 问题

我们正在编写使用回调函数的代码，但是担心小型函数在代码中大肆泛滥，程序的控制流会因此而失控。我们希望能有某种方法使代码看起来更像一般的过程式步骤。

7.11.2 解决方案

我们可以通过生成器和协程将回调函数内联到一个函数中。为了说明，假设有一个函数会按照下面的方式调用回调函数（参见7.10节）：

```
def

apply_async(func, args, *, callback):
    # Compute the result

    result = func(*args)

    # Invoke the callback with the result

    callback(result)
```

现在看看接下来的支持代码，这里涉及一个Async类和inlined_async装饰器：

```
from queue import


Queue
from functools import


wraps

class Async


:
    def


__init__(self, func, args):
        self.func = func
        self.args = args

def


inlined_async(func):
    @wraps(func)
    def


wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while
```

```
True:
            result = result_queue.get()
            try


:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue
            except StopIteration


:
                break




    return


wrapper
```

这两段代码允许我们通过yield语句将回调函数变为内联式的，示例如下：

```
def


add(x, y):
    return


x + y

@inlined_async
def


test():
    r = yield
```

```
Async(add, (2, 3))
    print

(r)
    r = yield

Async(add, ('hello', 'world'))
    print

(r)
    for

n in

range(10):
        r = yield

Async(add, (n, n))
        print

(r)
    print

('Goodbye')
```

如果调用test()，会得到这样的输出结果：

```
5
helloworld
0
2
```

```
4
6
8
10
12
14
16
18
Goodbye
```

除了那个特殊的装饰器和对yield的使用之外，我们会发现代码中根本就没有出现回调函数（它们只是隐藏在幕后了）。

7.11.3 讨论

本节将真正考验一下读者对回调函数、生成器以及程序控制流方面的掌控情况。

首先，在涉及回调函数的代码中，问题的关键就在于当前的计算会被挂起，然后在稍后某个时刻再得到恢复。当计算得到恢复时，回调函数将得以继续处理执行。示例中的apply_async()函数对执行回调函数的关键部分做了简单的说明，尽管在现实世界中这会复杂得多（涉及线程、进程、事件处理例程等）。

将计算挂起之后再恢复，这个思想非常自然地同生成器函数对应了起来。具体来说就是yield操作

使得生成器函数产生出一个值然后就挂起，后续调用生成器的__next__()或者send()方法会使得它再次启动。

鉴于此，本节的核心就在inline_async()装饰器函数中。关键点就是对于生成器函数的所有yield语句装饰器都会逐条进行跟踪，一次一个。为了做到这点，我们创建了一个队列用来保存结果，初始时用None来填充。之后通过循环将结果从队列中取出，然后发送给生成器，这样就会产生下一次yield，此时就会接收到Async的实例。然后在循环中查找函数和参数，开始异步计算apply_async()。但是，这个过程中最为隐蔽的部分就在于这里没有使用普通的回调函数，回调过程被设定到队列的put()方法中了。

此时应该可以精确描述到底都发生了些什么。主循环会迅速回到顶层，并在队列中执行一个get()操作。如果有数据存在，那它就一定是由put()回调产生的结果。如果什么都没有，操作就会阻塞，等待之后某个时刻会有结果到来。至于结果要如何产生，这取决于apply_async()函数的实现。

如果对这些疯狂的东西能否正常工作抱有怀疑，可以结合多进程库让异步操作在单独的进程中执行，以此测试该方案：

```
if


__name__ == '__main__':
    import multiprocessing



    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function



    test()
```

我们会发现这个方案的确能正常工作，但是要理清这其中的控制流程可能需要喝掉不少咖啡了。

将精巧的控制流隐藏在生成器函数之后，这种做法可以在标准库以及第三方包中找到。比如说，contextlib模块中的@contextmanager装饰器也使用了类似的令人费解的技巧，将上下文管理器的入口点和出口点通过一个yield语句粘合在了一起。著名的Twisted库（http://twistedmatrix.com ）中也有着类似的内联回调技巧。

7.12 访问定义在闭包内的变量

7.12.1 问题

我们希望通过函数来扩展闭包，使得在闭包内层定义的变量可以被访问和修改。

7.12.2 解决方案

一般来说，在闭包内层定义的变量对于外界来说完全是隔离的。但是，可以通过编写存取函数（accessor function，即getter/setter方法）并将它们作为函数属性附加到闭包上来提供对内层变量的访问支持。示例如下：

```
def

sample():
    n = 0
    # Closure function

    def

func():
        print
```

```
('n=', n)

    # Accessor methods for n



    def


get_n():
        return


n

    def


set_n(value):
        nonlocal n
        n = value

    # Attach as function attributes



    func.get_n = get_n
    func.set_n = set_n
    return


 func
```

下面是使用这份代码的示例:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
```

```
n= 10
>>> f.get_n()
10
>>>
```

7.12.3 讨论

这里主要用到了两个特性使得本节讨论的技术得以成功实施。首先，nonlocal声明使得编写函数来修改内层变量成为可能。其次，函数属性能够将存取函数以直接的方式附加到闭包函数上，它们工作起来很像实例的方法（尽管这里并没有涉及类）。

对本节提到的技术稍作扩展就可以让闭包模拟成类实例。我们所要做的就是将内层函数拷贝到一个实例的字典中然后将它返回。示例如下：

```
import sys



class ClosureInstance


:
    def


__init__(self, locals=None):
        if
```

```
locals is

None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables


        self.__dict__.update((key,value) for

key, value in

locals.items()
                             if

callable(value) )

    # Redirect special methods


    def

__len__(self):
        return

self.__dict__['__len__']()

# Example use


def

Stack():
```

```
    items = []

    def


push(item):
        items.append(item)

    def


pop():
        return


items.pop()

    def


__len__():
        return


len(items)
    return


ClosureInstance()
```

下面的交互式会话说明了这种方法确实能完成任务：

```
>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
```

```
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
10
>>>
```

有趣的是，这份代码运行起来比使用一个普通的类定义要稍微快一些。比如，我们可能会用下面这个类来做对比测试：

```
class Stack2


:
    def


__init__(self):
        self.items = []

    def


push(self, item):
        self.items.append(item)

    def


pop(self):
        return


self.items.pop()
```

```
    def


__len__(self):
        return


len(self.items)
```

如果进行对比测试，将得到类似如下的结果：

```
>>> from timeit import


timeit
>>> # Test involving closures




>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class




>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

我们可以看到，采用闭包的版本要快大约8%。测试中的大部分时间都花在对实例变量的直接访问上，闭包要更快一些，这是因为不用涉及额外的self变量。

Raymond Herringer在这个思路的基础上设计出了一种更加“恐怖”的变种。但是，在自己的代码中应该对这种奇技淫巧持谨慎的态度。请注意，相比一个真正的类，这种方法是相当怪异的。比如，像继承、属性、描述符或者类方法这样的主要特性在这种方法中都是无法使用的。我们还需要玩一些花招才能让特殊方法正常工作（比如，参考ClosureInstance中对__len__()的实现）。

最后，这么做会使得阅读你代码的人犯糊涂。他们会想知道这么做看起来和一个普通的类定义相比有什么区别（当然了，他们也想知道为什么这么做会运行的更快一些）。尽管如此，这仍然是个有趣的例子，它告诉我们对闭包内部提供访问机制能够实现出什么样的功能。

从全局的角度考虑，为闭包增加方法可能会有着更多的实际用途，比如我们想重置内部状态、刷新缓冲区、清除缓存或者实现某种形式的反馈机制（feedback mechanism）。

第8章　类与对象

本章的重点是为大家介绍一些与类定义相关的常见编程模式。主题包括让对象支持常见的Python特性、特殊方法的使用、封装、继承、内存管理以及一些有用的设计模式。

8.1 修改实例的字符串表示

8.1.1 问题

我们想修改打印实例所产生的输出，使输出结果能更有意义。

8.1.2 解决方案

要修改实例的字符串表示，可以通过定义__str__()和__repr__()方法来实现。示例如下：

```
class Pair


:
    def


__init__(self, x, y):
        self.x = x
        self.y = y
    def


__repr__(self):
        return


'Pair({0.x!r}, {0.y!r})'.format(self)
    def
```

```
__str__(self):
        return


'({0.x!s}, {0.y!s})'.format(self)
```

特殊方法__repr__()返回的是实例的代码表示（code representation），通常可以用它返回的字符串文本来重新创建这个实例[1]。内建的repr()函数可以用来返回这个字符串，当缺少交互式解释环境时可用它来检查实例的值。特殊方法__str__()将实例转换为一个字符串，这也是由str()和print()函数所产生的输出。示例如下：

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4)         # __repr__() output
>>> print


(p)
(3, 4)         # __str__() output
>>>
```

本节给出的实现中也展示了在进行格式化输出时应该如何使用不同的字符串表示。尤其是，特殊的格式化代码!r表示应该使用__repr__()的输出，而不是默认的__str__()。我们可以在前文给出的Pair类上做做实验：

```
>>> p = Pair(3, 4)
>>> print


('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print


('p is {0}'.format(p))
p is (3, 4)
>>>
```

8.1.3　讨论

定义__repr__()和__str__()通常被认为是好的编程实践，因为这么做可以简化调试过程和实例的输出。比方说，我们只用通过打印实例，程序员就能了解到更多有关这个实例内容的有用信息。

对于__repr__()，标准的做法是让它产生的字符串文本能够满足eval(repr(x)) == x。如果不可能办到或者说不希望有这种行为，那么通常就让它产生一段有帮助意义的文本，并且以<和>括起来。示例如下：

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

如果没有定义__str__()，那么就用__repr__()的输出当做备份。

解决方案中对format()函数的使用看起来似乎有点意思。格式化代码{0.x}用来指代参数0的x属性。因此在下面的函数中，0实际上就代表实例self：

```
def

__repr__(self):
    return

'Pair({0.x!r}, {0.y!r})'.format(self)
```

这个实现还可以有另外一种方式，可以使用%操作符和下面的代码来完成：

```
def

__repr__(self):
    return

'Pair(%r, %r)' % (self.x, self.y)
```

8.2　自定义字符串的输出格式

8.2.1　问题

我们想让对象通过format()函数和字符串方法来支持自定义的输出格式。

8.2.2　解决方案

要自定义字符串的输出格式，可以在类中定义__format__()方法。示例如下：

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
    }

class Date



:
    def



__init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def
```

```
__format__(self, code):
        if


code == '':
            code = 'ymd'
        fmt = _formats[code]
        return


fmt.format(d=self)
```

Date类的实例现在可以支持如下的格式化操作了：

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

8.2.3 讨论

__format__()方法在Python的字符串格式化功能中提供了一个钩子。需要重点强调的是，对格式化代码的解释完全取决于类本身。因此，格式化代码几乎可以为任何形式。举例来说，考虑下面的

datetime模块的示例：

```
>>> from datetime import


date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d,'%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

对于内建类型来说，有一些标准的格式化转换形式。请参阅string模块的文档（http://docs.python.org/3/library/string.html ）以获得正式的规范。

8.3　让对象支持上下文管理协议

8.3.1　问题

我们想让对象支持上下文管理协议（context-management protocol，通过with语句触发）。

8.3.2　解决方案

要让对象能够兼容with语句，需要实现__enter__()和__exit__()方法。比方说，考虑下面这个表示网络连接的类：

```
from socket import


socket, AF_INET, SOCK_STREAM

class LazyConnection


:
    def


__init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None
```

```
    def


__enter__(self):
        if


self.sock is not


None:
            raise RuntimeError


('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return


self.sock

    def


__exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

这个类的核心功能就是表示一条网络连接，但是实际上在初始状态下它并不会做任何事情（比如，它并不会建立一条连接）。相反，网络连接是通过with语句来建立和关闭的（这正是上下文管理的基本需求）。示例如下：

```
from functools import
```

```
partial

conn = LazyConnection(('www.python.org', 80))
# Connection closed



with


conn as


s:
    # conn.__enter__() executes: connection open



    s.send(b'GET /index.html HTTP/1.0\r\n


')
    s.send(b'Host: www.python.org\r\n


')
    s.send(b'\r\n


')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
```

8.3.3 讨论

要编写一个上下文管理器，其背后的主要原则

就是我们编写的代码需要包含在由with语句定义的代码块中。当遇到with语句时，__enter__()方法首先被触发执行。__enter__()的返回值（如果有的话）被放置在由as限定的变量当中。之后开始执行with代码块中的语句。最后，__exit__()方法被触发来执行清理工作。

这种形式的控制流与with语句块中发生了什么情况是没有关联的，出现异常时也是如此。实际上，__exit__()方法的三个参数就包含了异常类型、值和对挂起异常的追溯（如果出现异常的话）。__exit__()方法可以选择以某种方式来使用异常信息，或者什么也不干直接忽略它并返回None作为结果。如果__exit__()返回True，异常就会被清理干净，好像什么都没发生过一样，而程序也会立刻继续执行with语句块之后的代码。

这项技术有一个微妙的地方，那就是LazyConnection类是否可以通过多个with语句以嵌套的方式使用socket连接。正如我们给出的代码那样，一次只允许创建一条单独的socket连接。当socket已经在使用时，如果尝试重复使用with语句就会产生异常。我们可以对这个实现稍做修改来绕过这个限制，示例如下：

```
from socket import
```

```
socket, AF_INET, SOCK_STREAM

class LazyConnection


:
    def


__init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.connections = []

    def


__enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return


sock

    def


__exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Example use



from functools import


partial

conn = LazyConnection(('www.python.org', 80))
```

```
with

conn as

s1:
     ...
     with

conn as

s2:
          ...
          # s1 and s2 are independent sockets
```

在第二个版本中，LazyConnection成了一个专门生产网络连接的工厂类。在内部实现中，我们把一个列表当成栈使用来保存连接。每当__enter__()执行时，由它产生一个新的连接并添加到栈中。而__exit__()方法只是简单地将最近加入的那个连接从栈中弹出并关闭它。这个修改很微不足道，但是这样就可以允许用嵌套式的with语句一次创建出多个连接了。

上下文管理器最常用在需要管理类似文件、网络连接和锁这样的资源的程序中。这些资源的关键点在于它们必须显式地进行关闭或释放才能正确工

作。例如，如果获得了一个锁，之后就必须确保要释放它，否则就会有死锁的风险。通过实现__enter__()和__exit__()，并且利用with语句来触发，这类问题就可以很容易地避免了。因为__exit__()方法中的清理代码无论如何都会保证运行的。

有关上下文管理器的另一种构想可以在contextmanager模块中找到，请参阅9.22节。本节示例的线程安全版本可以在12.6节中找到。

8.4　当创建大量实例时如何节省内存

8.4.1　问题

我们的程序创建了大量的（比如百万级）实例，为此占用了大量的内存。

8.4.2　解决方案

对于那些主要用作简单数据结构的类，通常可以在类定义中增加__slot__属性，以此来大量减少对内存的使用。示例如下：

```
class Date


:
    __slots__ = ['year', 'month', 'day']
    def


__init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

当定义了__slots__属性时，Python就会针对实例采用一种更加紧凑的内部表示。不再让每个实例都创建一个__dict__字典，现在的实例是围绕着一个固定长度的小型数组来构建的，这和一个元组或者列表很相似。在__slots__中列出的属性名会在内部映射到这个数组的特定索引上。使用__slots__带来的副作用是我们没法再对实例添加任何新的属性了——我们被限制为只允许使用__slots__中列出的那些属性名。

8.4.3　讨论

使用__slots__节省下来的内存根据创建的实例数量以及保存的属性类型而有所不同。但是，一般来说使用的内存量相当与将数据保存在元组中。为了有一个直观的感受，我们举个例子：在64位版本的Python中，不使用__slots__保存一个单独的Date实例，则需要占用428字节的内存。如果定义了__slots__，内存用量将下降到156字节。在一个需要同时处理大量Date实例的程序中，这将显著减少总的内存用量。

尽管__slots__看起来似乎是一个非常有用的特性，但是在大部分代码中都应该尽量别使用它。Python中有许多部分都依赖于传统的基于字典的实现。此外，定义了__slots__属性的类不支持某些特

定的功能，比如多重继承。就大部分情况而言，我们应该只针对那些在程序中被当做数据结构而频繁使用的类上采用__slots__技法（例如，如果你的程序创建了上百万个特定的类实例）。

关于__slots__有一个常见的误解，那就是这是一种封装工具，可以阻止用户为实例添加新的属性。尽管这的确是使用__slots__所带来的副作用，但这绝不是使用__slots__的原本意图。相反，人们一直以来都把__slots__当做一种优化工具。

8.5 将名称封装到类中

8.5.1 问题

我们想将“私有”数据封装到类的实例上，但是又需要考虑到Python缺乏对属性的访问控制问题。

8.5.2 解决方案

与其依赖语言特性来封装数据，Python程序员们更期望通过特定的命名规则来表达出对数据和方法的用途。第一个规则是任何以单下划线（_）开头的名字应该总是被认为只属于内部实现。比如：

```
class A

:
    def

__init__(self):
        self._internal = 0         # An internal attribute

        self.public = 1         # A public attribute
```

```
    def
public_method(self):
        '''
        A public method
        '''
        ...
    def
_internal_method(self):
        ...
```

Python本身并不会阻止其他人访问内部名称。但是如果有人这么做了，则被认为是粗鲁的，而且可能导致产生出脆弱不堪的代码。应该要提到的是，以下划线打头的标识也可用于模块名称和模块级的函数中。比如，如果见到有模块名以下划线打头（例如，_socket），那么它就属于内部实现。同样地，模块级的函数比如sys._getframe()使用起来也要格外小心。

我们应该在类定义中也见到过以双下划线（__）打头的名称。例如：

```
class B


:
    def


__init__(self):
        self.__private = 0
    def


__private_method(self):
        ...
    def


public_method(self):
        ...
        self.__private_method()
        ...
```

以双下划线打头的名称会导致出现名称重整（name mangling）的行为。具体来说就是上面这个类中的私有属性会被分别重命名为_B__private和_B__private_method。此时你可能会问，类似这样的名称重整其目的何在？答案就是为了继承——这样的属性不能通过继承而覆盖。示例如下：

```
class C


(B):
    def
```

```
__init__(self):
        super().__init__()
        self.__private = 1          # Does not override B.__priva

    # Does not override B.__private_method()

    def

__private_method(self):
        ...
```

这里，私有名称__private和__private_method会被重命名为_C__private和_C__ private_method，这和基类B中的重整名称不同。

8.5.3 讨论

“私有”属性存在两种不同的命名规则（单下划线和双下划线），这一事实引出了一个显而易见的问题：应该使用哪种风格？对于大部分代码而言，我们应该让非公有名称以单下划线开头。但是，如果我们知道代码中会涉及子类化处理，而且有些内部属性应该对子类进行隐藏，那么此时就应该使用双下划线开头。

此外还应该指出的是，有时候可能想定义一个变量，但是名称可能会和保留字产生冲突。基于此，应该在名称最后加上一个单下划线以示区别。比如：

```
lambda_ = 2.0     # Trailing _ to avoid clash with lambda keyword
```

这里不采用以下划线开头的原因是避免在使用意图上发生混淆（例如，如果采用下划线开头的形式，那么可能会被解释为这么做是为了避免名称冲突，而不是作为私有数据的标志）。在名称尾部加一个单下划线就能解决这个问题。

8.6 创建可管理的属性

8.6.1 问题

在对实例属性的获取和设定上，我们希望增加一些额外的处理过程（比如类型检查或者验证）。

8.6.2 解决方案

要自定义对属性的访问，一种简单的方式是将其定义为property [2] 。比如说，下面的代码定义了一个property，增加了对属性的类型检查：

```
class Person


:
    def


__init__(self, first_name):
        self.first_name = first_name

    # Getter function



    @property
    def
```

```
first_name(self):
        return


self._first_name

    # Setter function



    @first_name.setter
    def


first_name(self, value):
        if not


isinstance(value, str):
            raise TypeError


('Expected a string')
        self._first_name = value

    # Deleter function (optional)



    @first_name.deleter
    def


first_name(self):
        raise AttributeError


("Can't delete attribute")
```

在上述代码中，一共有三个互相关联的方法，它们必须有着相同的名称。第一个方法是一个getter

函数，并且将first_name定义为了property属性。其他两个方法将可选的setter和deleter函数附加到了first_name属性上。需要重点强调的是，除非first_name已经通过@property的方式定义为了property属性，否则是不能定义@first_name.setter和@first_name.deleter装饰器的。

property的重要特性就是它看起来就像一个普通的属性，但是根据访问它的不同方式，会自动触发getter、setter以及deleter方法。示例如下：

```
>>> a = Person('Guido')
>>> a.first_name        # Calls the getter



'Guido'
>>> a.first_name = 42     # Calls the setter



Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "prop.py", line 14, in first_name
    raise TypeError


('Expected a string')
TypeError: Expected a string
>>> del


a.first_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>
```

当我们实现一个property时，底层的数据（如果有的话）仍然需要被保存到某个地方。因此在get和set方法中，可以看到我们是直接对_first_name进行操作的，这就是数据实际保存的地方。此外，你可能会问为什么在__init__()方法中设定的是self.first_name而不是self._first_name呢？在这个例子中，property的全部意义就在于我们设置属性时可以执行类型检查。因此，很有可能你想让这种类型检查在初始化的时候也可以进行。因此，在__init__()中设置self.first_name，实际上会调用到setter方法（因此就会跳过self.first_name而去访问self._first_name）。

对于已经存在的get和set方法，同样也可以将它们定义为property。示例如下：

```
class Person

:
    def

__init__(self, first_name):
        self.set_first_name(first_name)

    # Getter function
```

```
    def

get_first_name(self):
        return

self._first_name

    # Setter function

    def

set_first_name(self, value):
        if not

isinstance(value, str):
            raise TypeError

('Expected a string')
        self._first_name = value

    # Deleter function (optional)

    def

del_first_name(self):
        raise AttributeError

("Can't delete attribute")

    # Make a property from existing get/set methods
```

```
    name = property(get_first_name, set_first_name, del_first_nar
```

8.6.3 讨论

property属性实际上就是把一系列的方法绑定到一起。如果检查类的property属性，就会发现property自身所持有的属性fget、fset和fdel所代表的原始方法。示例如下：

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

一般来说我们不会直接去调用fget或者fset，但是当我们访问property属性时会自动触发对这些方法的调用。

只有当确实需要在访问属性时完成一些额外的处理任务时，才应该使用property。有时候Java程序员会觉得所有的访问都需要通过getter和setter来处理，那么他们的代码就应该是下面这个样子：

```
class Person


:
    def


__init__(self, first_name):
        self.first_name = name
    @property
    def


first_name(self):
        return


self._first_name
    @first_name.setter
    def


first_name(self, value):
        self._first_name = value
```

如果property并不会完成任何额外的处理任务，就不要把代码写成上面这个样子。第一，这么做会使得代码变得更加啰嗦，对其他人来说也比较困惑。第二，这么做会让程序变慢很多。最后，这么做不会给设计带来真正的好处。特别是如果稍后决定要对某个普通的属性增加额外的处理步骤时，可以在不修改已有代码的情况下将这个属性提升为一个property。这是因为代码中访问一个属性的语法并不会改变（即，访问普通属性和访问property属性的代码写法是一样的）。

property也可以用来定义需要计算的属性。这类属性并不会实际保存起来，而是根据需要完成计算。示例如下：

```
import math

class Circle

:
    def

__init__(self, radius):
        self.radius = radius
    @property
    def

area(self):
        return

math.pi * self.radius ** 2
    @property
    def

perimeter(self):
        return

2 * math.pi * self.radius
```

这里对property的使用使得实例的接口变得非

常统一，radius、area以及perimeter都能够简单地以属性的形式进行访问，而不必将属性和方法调用混在一起使用了。示例如下：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area             # 注意这里没有()



50.26548245743669
>>> c.perimeter         # 这里也没有()



25.132741228718345
>>>
```

尽管property带来了优雅的编程接口，但有时候我们还是希望能够直接使用getter和setter函数。比如说：

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

这种情况常常会出现在当Python代码需要被集成到一个更为庞大的系统基础设施或者程序的时

候。比方说，也许有一个Python类需要根据远程过程调用（RPC）或者分布式对象插入到一个大型的分布式系统中。在这种情况下，直接显式地采用get/set方法（作为普通的方法调用）要比通过property来隐式调用这类函数更加方便和简单。

最后但也同样重要的是，不要编写那种定义了大量重复性property的代码。示例如下：

```
class Person


:
    def


__init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def


first_name(self):
        return


self._first_name

    @first_name.setter
    def


first_name(self, value):
        if not
```

```
isinstance(value, str):
            raise TypeError


('Expected a string')
        self._first_name = value

    # Repeated property code, but for a different name (bad!)



    @property
    def


last_name(self):
        return


self._last_name

    @last_name.setter
    def


last_name(self, value):
        if not


isinstance(value, str):
            raise TypeError


('Expected a string')
        self._last_name = value
```

重复的代码会导致代码膨胀，容易出错，而且代码也十分丑陋。事实证明，利用描述符或者闭包能够更好地完成同样的任务，具体请参见8.9节和

9.21节。

8.7　调用父类中的方法

8.7.1　问题

我们想调用一个父类中的方法，这个方法在子类中已经被覆盖了。

8.7.2　解决方案

要调用父类（或称超类）中的方法，可以使用super()函数完成。示例如下：

```
class A

:
    def

spam(self):
        print

('A.spam')

class B

(A):
    def
```

```
spam(self):
        print

('B.spam')
        super().spam()     # Call parent spam()
```

super()函数的一种常见用途是调用父类的__init__()方法，确保父类被正确地初始化了：

```
class A

:
    def

__init__(self):
        self.x = 0

class B

(A):
    def

__init__(self):
        super().__init__()
        self.y = 1
```

另一种常见用途是当覆盖了Python中的特殊方

法时，示例如下：

```
class Proxy


:
    def


__init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj



    def


__getattr__(self, name):
        return


getattr(self._obj, name)

    # Delegate attribute assignment



    def


__setattr__(self, name, value):
        if


name.startswith('_'):
            super().__setattr__(name, value)     # Call original



        else
```

```
:
            setattr(self._obj, name, value)
```

在上述代码中，__setattr__()的实现里包含了对名称的检查。如果名称是以一个下划线（_）开头的，它就通过super()去调用原始的__setattr__()实现。否则，就转而对内部持有的对象self._obj进行操作。这看起来有点意思，但是super()即使在没有显式列出基类的情况下也是可以工作的。

8.7.3 讨论

如何正确使用super()函数，这实际上是人们在Python中理解的最差的知识点之一。偶尔我们会看到一些代码直接调用父类中的方法，就像这样：

```
class Base

:
    def

__init__(self):
        print

('Base.__init__')
```

```
class A


(Base):
    def


__init__(self):
        Base.__init__(self)
        print


('A.__init__')
```

尽管对于大部分代码来说这么做都“行得通”，但是在涉及多重继承的代码里，就会导致出现奇怪的麻烦。比如，考虑下面这个例子：

```
class Base


:
    def


__init__(self):
        print


('Base.__init__')

class A


(Base):
    def
```

```
__init__(self):
        Base.__init__(self)
        print


('A.__init__')

class B


(Base):
    def


__init__(self):
        Base.__init__(self)
        print


('B.__init__')

class C


(A,B):
    def


__init__(self):
        A.__init__(self)
        B.__init__(self)
        print


('C.__init__')
```

如果运行上面的代码，会发现Base.__init__()方法被调用了两次。如下所示：

```
>>> c = C()
```

```
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

也许调用两次Base.__init__()并没什么害处，但是也可能刚好相反。如果从另一方面考虑，将代码修改为使用super()，那么一切就都能正常工作了：

```
class Base


:
    def


__init__(self):
        print


('Base.__init__')

class A


(Base):
    def


__init__(self):
        super().__init__()
        print


('A.__init__')
```

```
class B


(Base):
    def


__init__(self):
        super().__init__()
        print


('B.__init__')

class C


(A,B):
    def


__init__(self):
        super().__init__() # Only one call to super() here



        print


('C.__init__')
```

当使用这个新版的代码时，就会发现每个__init__()方法都只调用了一次：

```
>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>
```

要理解其中的缘由，我们需要退一步，先讨论一下Python是如何实现继承的。针对每一个定义的类，Python都会计算出一个称为方法解析顺序（MRO）的列表[3]。MRO列表只是简单地对所有的基类进行线性排列。示例如下：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)
>>>
```

要实现继承，Python从MRO列表中最左边的类开始，从左到右依次查找，直到找到待查的属性时为止。

而MRO列表本身又是如何确定的呢？这里用到了一种称为C3线性化处理（C3 Linearization）的技术。为了不陷入到艰深的数学理论中，简单来说这就是针对父类的一种归并排序，它需要满足3个约束：

- 先检查子类再检查父类；
- 有多个父类时，按照MRO列表的顺序依次

检查;

- 如果下一个待选的类出现了两个合法的选择，那么就从第一个父类中选取。

老实说，所有需要的知道的就是MRO列表中对类的排序几乎适用于任何定义的类层次结构（class hierarchy）。

当使用super()函数时，Python会继续从MRO中的下一个类开始搜索。只要每一个重新定义过的方法（也就是覆盖方法）都使用了super()，并且只调用了它一次，那么控制流最终就可以遍历整个MRO列表，并且让每个方法只会被调用一次。这就是为什么在第二个例子中Base.__init__()不会被调用两次的原因。

关于 super()，一个有些令人惊讶的方面是，它并不是一定要关联到某个类的直接父类上，甚至可以在没有直接父类的类中使用它。例如，考虑下面这个类：

```
class A

:
    def
```

```
spam(self):
        print


('A.spam')




super().spam()
```

如果试着使用这个类，会发现这完全行不通：

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

但是，如果把这个类用于多重继承时看看会发生什么：

```
>>> class B


:
...     def


spam(self):
...         print


('B.spam')
```

```
...

>>> class C

(A,B):
...     pass

...

>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

这里我们会发现在类A中使用的super().spam()实际上居然调用到了类B中的spam()方法——B和A是完全不相关的！这一切都可以用类C的MRO列表来解释：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
>>>
```

我们常常会在定义混合类（mixin class）时以这种方式使用super()。请参见8.13和8.18节。

但是，由于super()可能会调用到我们不希望调用的方法，那么这里有一些应该遵守的基本准则。首先，确保在继承体系中所有同名的方法都有可兼容的调用签名（即，参数数量相同，参数名称也相同）。如果super()尝试去调用非直接父类的方法，那么这就可以确保不会遇到麻烦。其次，确保最顶层的类实现了这个方法通常是个好主意。这样沿着MRO列表展开的查询链会因为最终找到了实际的方法而终止。

在Python社区中，关于super()的使用有时候会成为争论的焦点。但是，公平地说，我们应该在现代的代码中使用它。Raymond Hettinger在博客中写过一篇题为“Python’s super() considered Super!”的文章，文章中列举了更多的示例和理由来说明为什么super()会是超级有用的工具 [4] 。

8.8　在子类中扩展属性

8.8.1　问题

我们想在子类中扩展某个属性的功能，而这个属性是在父类中定义的。

8.8.2　解决方案

考虑如下的代码，这里我们定义了一个属性name：

```
class Person

:
    def

__init__(self, name):


self.name = name



# Getter function
```

```
@property
    def

name(self):
        return

self._name


# Setter function



@name.setter
    def

name(self, value):
        if not

isinstance(value, str):
            raise TypeError

('Expected a string')


self._name = value


# Deleter function
```

```
@name.deleter
    def

name(self):
        raise AttributeError

("Can't delete attribute")
```

下面我们从Person类中继承，然后在子类中扩展name属性的功能：

```
class SubPerson

(Person):


@property
    def

name(self):
        print

('Getting name')
        return

super().name
```

```
@name.setter
    def

name(self, value):
        print

('Setting name to', value)


super(SubPerson, SubPerson).name.__set__(self, value)



@name.deleter
    def

name(self):
        print

('Deleting name')


super(SubPerson, SubPerson).name.__delete__(self)
```

下面是使用这个新类的示例:

```
>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
```

```
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
        raise TypeError


('Expected a string')
  TypeError: Expected a string
  >>>
```

如果只想扩展属性中的其中一个方法，可以使用下面的代码实现：

```
class SubPerson


(Person):



@Person.name.getter
    def


name(self):
        print


('Getting name')
        return


super().name
```

或者，如果只想扩展setter，可以这样：

```
class SubPerson


(Person):



@Person.name.setter
    def


name(self, value):
        print


('Setting name to', value)



super(SubPerson, SubPerson).name.__set__(self, value)
```

8.8.3 讨论

在子类中扩展属性会引入一些非常微妙的问题，因为属性其实是被定义为getter、setter和deleter方法的集合，而不仅仅只是单独的方法。因此，当我们扩展一个属性时，需要弄清楚是要重新定义所有的方法还是只针对其中一个方法做扩展。

在第一个例子中，所有的属性方法都被重新定

义了。在每个方法中，我们利用super()函数来调用之前的实现。在setter函数中，对super(SubPerson, SubPerson).name.__ set__(self, value)的调用并不是错误，下面我们来解释一下。为了调用到setter之前的实现，需要把控制流传递到之前定义的name属性的__set__()方法中去。但是，唯一能调用到这个方法的方式就是以类变量而不是实例变量的方式去访问。这正是super(SubPerson, SubPerson)操作所完成的任务。

如果只想重新定义其中的一个方法，只使用@property是不够的。例如，下面这样的代码是无法工作的：

```
class SubPerson

(Person):


@property             # Doesn't work
    def

 name(self):
        print

('Getting name')
        return

 super().name
```

```
```

如果试着使用这份代码，就会发现setter函数完全消失不见了：

```
>>>

 s = SubPerson('Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 5, in __init__

self.name = name
AttributeError: can't set attribute
>>>
```

相反，我们应该将代码修改为解决方案中的那样：

```
class SubPerson

(Person):

@Person.getter
    def

 name(self):
```

```
        print

('Getting name')
        return

 super().name
```

当这么做之后，所有之前定义过的属性方法都会被拷贝过来，而getter函数则会被替换掉。现在可以按照预期的方式工作了：

```
>>>

 s = SubPerson('Guido')
>>>

 s.name
Getting name
'Guido'
>>>

 s.name = 'Larry'
>>>

 s.name
Getting name
'Larry'
>>>

 s.name = 42
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
    raise


 TypeError('Expected a string')
TypeError: Expected a string
>>>
```

在这个特定的解决方案中，我们没法以更加一般化的名称来替换硬编码的类名Person。如果不清楚哪个基类定义了属性，则应该采用这样的方案：重新定义所有的属性方法，并利用super()来调用之前的实现。

值得一提的是，本节展示的第一个技术同样也可以用来扩展描述符（见8.9节）。示例如下：

```
# A descriptor


class String


:
    def


 __init__(self, name):


self.name = name
```

```
    def

 __get__(self, instance, cls):
        if

instance is None:
            return

 self
        return

 instance.__dict__[self.name]

    def

 __set__(self, instance, value):
        if not

 isinstance(value, str):
            raise TypeError

('Expected a string')


instance.__dict__[self.name] = value

# A class with a descriptor


class Person

:
```

```
name = String('name')
    def

 __init__(self, name):


self.name = name

# Extending a descriptor with a property


class SubPerson

(Person):


@property
    def

 name(self):
        print

('Getting name')
        return

 super().name



@name.setter
    def

 name(self, value):
        print
```

```
('Setting name to', value)

super(SubPerson, SubPerson).name.__set__(self, value)

@name.deleter
    def

 name(self):
        print

('Deleting name')

super(SubPerson, SubPerson).name.__delete__(self)
```

最后要说的是，当读到这里的时候，我们应该会觉得在子类中重定义setter和deleter的工作多少得到了一些简化。虽然这里给出的解决方案仍然能够正常工作，但是在Python的问题报告页面中提到的这个bug（http://bugs.python.org/issue14965 ）可能会使得在未来的Python版本中产生出一种更加清晰的解决方案。

8.9　创建一种新形式的类属性或实例属性

8.9.1　问题

我们想创建一种新形式的实例属性，它可以拥有一些额外的功能，比如说类型检查。

8.9.2　解决方案

如果想创建一个新形式的实例属性，可以以描述符类的形式定义其功能。示例如下：

```
# Descriptor attribute for an integer type-checked attribute

class Integer

:
    def

 __init__(self, name):

self.name = name
```

```
    def

 __get__(self, instance, cls):
        if

 instance is None:
            return

 self
        else

:
            return

 instance.__dict__[self.name]

    def

 __set__(self, instance, value):
        if not

 isinstance(value, int):
            raise TypeError

('Expected an int')


instance.__dict__[self.name] = value

    def

 __delete__(self, instance):
        del
```

```
 instance.__dict__[self.name]
```

所谓的描述符就是以特殊方法__get__()、__set__()和__delete__()的形式实现了三个核心的属性访问操作（对应于get、set和delete）的类。这些方法通过接受类实例作为输入来工作。之后，底层的实例字典会根据需要适当地进行调整。

要使用一个描述符，我们把描述符的实例放置在类的定义中作为类变量来用。示例如下：

```
class Point


:
    x = Integer('x')
    y = Integer('y')
    def


 __init__(self, x, y):
        self.x = x
        self.y = y
```

当这么做时，所有针对描述符属性（即，这里的x或y）的访问都会被__get__()、__set__()和__delete__()方法所捕获。示例如下：

```
>>>
 p = Point(2, 3)
>>>
 p.x                # Calls Point.x__get__(p, Point)
2
>>>
 p.y = 5            # Calls Point.y.__set__(p, 5)
>>>
 p.x = 2.3          # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 12, in __set__
    raise TypeError
('Expected an int')
TypeError: Expected an int
>>>
```

每个描述符方法都会接受被操纵的实例作为输入。要执行所请求的操作，底层的实例字典（即__dict__属性）会根据需要适当地进行调整。描述

符的self.name属性会保存字典的键，通过这些键可以找到存储在实例字典中的实际数据。

8.9.3　讨论

对于大多数Python类的特性，描述符都提供了底层的魔法，包括@classmethod、@staticmethod、@property甚至__slots__。

通过定义一个描述符，我们可以在很底层的情况下捕获关键的实例操作（get、set、delete），并可以完全自定义这些操作的行为。这种能力非常强大，这也是那些编写高级程序库和框架的作者们所使用的最为重要的工具之一。

关于描述符，常容易困惑的地方就是它们只能在类的层次上定义，不能根据实例来产生。因此，下面这样的代码是无法工作的：

```
# Does NOT work

class Point

:
    def
```

```
__init__(self, x, y):
        self.x = Integer('x')     # No! Must be a class variable



        self.y = Integer('y')
        self.x = x
        self.y = y
```

此外，在实现__get__()方法时比想象中的还要复杂一些：

```
# Descriptor attribute for an integer type-checked attribute



class Integer


:
    ...
    def


__get__(self, instance, cls):
        if


instance is


None:
            return


self
        else
```

```
:
            return
instance.__dict__[self.name]
    ...
```

__get__()看起来多少有些复杂的原因在于实例变量和类变量之间是有区别的。如果是以类变量的形式访问描述符，参数instance应该设为None。在这种情况下，标准做法就是简单地返回描述符实例本身（尽管此时做任何类型的自定义处理也是允许的）。示例如下：

```
>>> p = Point(2,3)
>>> p.x         # Calls Point.x.__get__(p, Point)
2
>>> Point.x     # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

描述符常常会作为一个组件出现在大型的编程框架中，其中还会涉及装饰器或者元类。正因为如此，对描述符的使用可能隐藏得很深，几乎看不到痕迹。例如，下面是一些更加高级的基于描述符的

代码，其中还用到了类装饰器：

```
# Descriptor for a type-checked attribute



class Typed


:
    def


__init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type

    def


__get__(self, instance, cls):
        if


instance is


None:
            return


self
        else


:
            return


instance.__dict__[self.name]

    def
```

```
__set__(self, instance, value):
        if not

isinstance(value, self.expected_type):
            raise TypeError

('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value

    def

__delete__(self, instance):
        del

instance.__dict__[self.name]

# Class decorator that applies it to selected attributes

def

typeassert(**kwargs):
    def

decorate(cls):
        for

name, expected_type in

kwargs.items():
            # Attach a Typed descriptor to the class
```

```
            setattr(cls, name, Typed(name, expected_type))
        return


cls
    return


decorate

# Example use



@typeassert(name=str, shares=int, price=float)
class Stock


:
    def


__init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

最后，应该强调的是，如果只是想访问某个特定的类中的一种属性，并对此做定制化处理，那么最好不要编写描述符来实现。对于这个任务，用property属性方法来完成会更加简单（见8.6节）。在需要大量重用代码的情况下，描述符会更加有用（例如，我们希望在自己的代码中大量使用描述符提供的功能，或者将其作为库来使用）。

8.10 让属性具有惰性求值的能力

8.10.1 问题

我们想将一个只读的属性定义为property属性方法，只有在访问它时才参与计算。但是，一旦访问了该属性，我们希望把计算出的值缓存起来，不要每次访问它时都重新计算。

8.10.2 解决方案

定义一个惰性属性最有效的方式就是利用描述符类来完成，示例如下：

```
class lazyproperty

:
    def

__init__(self, func):
        self.func = func

    def

__get__(self, instance, cls):
        if
```

```
instance is


None:
            return


self
        else


:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return


value
```

要使用上述代码，可以像下面这样在某个类中使用它：

```
import math




class Circle


:
    def


__init__(self, radius):
        self.radius = radius

    @lazyproperty
    def
```

```
area(self):
        print

('Computing area')
        return

math.pi * self.radius ** 2

    @lazyproperty
    def

perimeter(self):
        print

('Computing perimeter')
        return

2 * math.pi * self.radius
```

下面的交互式会话说明了这是如何工作的：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
25.132741228718345
```

```
>>>
```

请注意，这里的“Computing area”和“Computing perimeter”只打印了一次。

8.10.3　讨论

在大部分情况下，让属性具有惰性求值能力的全部意义就在于提升程序性能。例如，除非确实需要用到这个属性，否则就可以避免进行无意义的计算。本节给出的解决方案正是应对于此，而且利用了描述符的微妙特性，使得能够以高效的方式来达成。

在8.9节中讲过，当把描述符放到类的定义体中时，访问它的属性会触发__get__()、__set__()和__delete__()方法得到执行。但是，如果一个描述符只定义了__get__()方法，则它的绑定关系比一般情况下要弱化很多（much weaker binding）。特别是，只有当被访问的属性不在底层的实例字典中时，__get__()方法才会得到调用。

示例中的lazyproperty类通过让__get__()方法以property属性相同的名称来保存计算出的值。这么做会让值保存在实例字典中，可以阻止该property

属性重复进行计算。仔细观察下面的示例就能发现这一点：

```
>>> c = Circle(4.0)
>>> # Get instance variables



>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward



>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore



>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again



>>> del


c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

本节讨论的技术有一个潜在的缺点，即，计算出的值在创建之后就变成可变的（mutable）了。示例如下：

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

如果需要考虑可变性的问题，可以使用另外一种方式实现，但执行效率会稍打折扣：

```
def


lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def


lazy(self):
        if


hasattr(self, name):
            return


getattr(self, name)
```

```
        else

:
            value = func(self)
            setattr(self, name, value)
            return

value
    return

lazy
```

如果使用这个版本的实现，就会发现set操作是不允许执行的。示例如下：

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

但是，这种方式的缺点就是所有的get操作都必须经由属性的getter函数来处理。这比直接在实例字典中查找相应的值要慢一些。

更多有关property和可管理属性的信息，请参见8.6节。描述符在8.9节已有详尽的讲解。

8.11 简化数据结构的初始化过程

8.11.1 问题

我们编写了许多类，把它们当做数据结构来用。但是我们厌倦了编写高度重复且样式相同的__init__()函数。

8.11.2 解决方案

通常我们可以将初始化数据结构的步骤归纳到一个单独的__init__()函数中，并将其定义在一个公共的基类中。示例如下：

```
class Structure

:
    # Class variable that specifies expected fields

    _fields= []
    def

__init__(self, *args):
        if

len(args) != len(self._fields):
```

```
            raise TypeError

('Expected {} arguments'.format(len(self._fields)))
        # Set the arguments

        for

name, value in

zip(self._fields, args):
            setattr(self, name, value)

    # Example class definitions

    if

__name__ == '__main__':
        class Stock

(Structure):
            _fields = ['name', 'shares', 'price']

        class Point

(Structure):
            _fields = ['x','y']

        class Circle

(Structure):
            _fields = ['radius']
            def
```

```
area(self):
                return


math.pi * self.radius ** 2
```

如果使用这些类，就会发现它们非常易于构建。示例如下：

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "structure.py", line 6, in __init__
    raise TypeError


('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

我们应该提供对关键字参数的支持，这里有几种设计上的选择。一种选择就是对关键字参数做映射，这样它们就只对应于定义在_fields中的属性名。示例如下：

```
class Structure


:
```

```
    _fields= []
    def

__init__(self, *args, **kwargs):
        if

len(args) > len(self._fields):
            raise TypeError

('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments


        for

name, value in

zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments


        for

name in

self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))

        # Check for any remaining unknown arguments


        if
```

```
kwargs:
            raise TypeError


('Invalid argument(s): {}'.format(','.join(kwargs)))

# Example use



if



__name__ == '__main__':
    class Stock


(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
```

另一种可能的选择是利用关键字参数来给类添加额外的属性，这些额外的属性是没有定义在_fields中的。示例如下：

```
class Structure


:
    # Class variable that specifies expected fields
```

```
    _fields= []
    def


__init__(self, *args, **kwargs):
        if


len(args) != len(self._fields):
            raise TypeError


('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments



        for


name, value in


zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)



        extra_args = kwargs.keys() - self._fields
        for


name in


extra_args:
            setattr(self, name, kwargs.pop(name))
        if


kwargs:
            raise TypeError
```

```
('Duplicate values for {}'.format(','.join(kwargs)))

# Example use



if



__name__ == '__main__':
    class Stock



(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')
```

8.11.3　讨论

如果要编写的程序中有大量小型的数据结构，那么定义一个通用型的__init__()方法会特别有用。相比于下面这样手动编写每个__init__()方法，这么做可使得代码量大大减少：

```
class Stock



:
    def



__init__(self, name, shares, price):
```

```
        self.name = name
        self.shares = shares
        self.price = price

class Point



:
    def



__init__(self, x, y):
        self.x = x
        self.y = y

class Circle



:
    def



__init__(self, radius):
        self.radius = radius
    def



area(self):
        return



math.pi * self.radius ** 2
```

我们给出的实现中，一个微妙之处在于使用了setattr()函数来设定属性值。与之相反的是，有人可能会倾向于直接访问实例字典。示例如下：

```
class Structure
```

```
:
    # Class variable that specifies expected fields


    _fields= []
    def


__init__(self, *args):
        if


len(args) != len(self._fields):
            raise TypeError


('Expected {} arguments'.format(len(self._fields)))

    # Set the arguments (alternate)



    self.__dict__.update(zip(self._fields,args))
```

尽管这么做行得通，但像这样假设子类的实现通常是不安全的。如果某个子类决定使用__slots__或者用property（也可以是描述符）包装了某个特定的属性，直接访问实例字典就会产生崩溃。我们给出的解决方案已经尽可能地做到通用，不会对子类的实现做任何假设。

这种技术的一个潜在缺点就是会影响到IDE（集成开发环境）的文档和帮助功能。如果用

户针对某个特定的类寻求帮助，那么所需的参数将不会以正常的形式来表述。示例如下：

```
>>> help(Stock)
Help on class Stock in module __main__:

class Stock(Structure)
...



| Methods inherited from Structure:
|
| __init__(self, *args, **kwargs)
|
...



>>>
```

这些问题可以通过在__init__()函数中强制施行类型签名来解决，相关内容请参阅9.16节。

应该指出的是，也可以采用所谓的“frame hack”技巧来实现自动化的实例变量初始化处理，只要编写一个功能函数即可。示例如下：

```
def


init_fromlocals(self):
    import sys
```

```
    locs = sys._getframe(1).f_locals
    for

k, v in

locs.items():
        if

k != 'self':
            setattr(self, k, v)
class Stock

:
    def

__init__(self, name, shares, price):
        init_fromlocals(self)
```

在这种方法中，函数init_fromlocals()利用sys._getframe()来获取调用方的局部变量。如果在__init__()方法中首先调用这个函数，那么获取到的局部变量就和传递给__init__()方法的参数是一致的，可以轻松用来设定属性。尽管这种方法可以避免在IDE中出现获取到不一致的调用签名问题，但比起解决方案中提供的方法要慢上50%，也需要程序员输入更多的代码，这种方法在幕后也做了更加复杂的操作。如果我们的代码不需要这种额外的能力，那么通常更简单的方案会更好。

8.12　定义一个接口或抽象基类

8.12.1　问题

我们想定义一个类作为接口或者是抽象基类，这样可以在此之上执行类型检查并确保在子类中实现特定的方法。

8.12.2　解决方案

要定义一个抽象基类，可以使用abc模块。示例如下：

```
from abc import


ABCMeta, abstractmethod

class IStream


(metaclass=ABCMeta):
    @abstractmethod
    def


read(self, maxbytes=-1):
        pass
```

```
    @abstractmethod
    def


write(self, data):
        pass
```

抽象基类的核心特征就是不能被直接实例化。例如，如果尝试这么做，会得到错误提示：

```
a = IStream()     # TypeError: Can't instantiate abstract class


                # IStream with abstract methods read, write
```

相反，抽象基类是用来给其他的类当做基类使用的，这些子类需要实现基类中要求的那些方法。示例如下：

```
class SocketStream


(IStream):
    def
```

```
read(self, maxbytes=-1):
        ...
    def


write(self, data):
        ...
```

抽象基类的主要用途是强制规定所需的编程接口。例如，一种看待IStream基类的方式就是在高层次上指定一个接口规范，使其允许读取和写入数据。显式检查这个接口的代码可以写成如下形式：

```
def


serialize(obj, stream):
    if not


isinstance(stream, IStream):
        raise TypeError


('Expected an IStream')
    ...
```

我们可能会认为这种形式的类型检查只有在子类化抽象基类（ABC）时才能工作，但是抽象基类也允许其他的类向其注册，然后实现所需的接口。例如，我们可以这样做：

```
import io



# Register the built-in I/O classes as supporting our interface



IStream.register(io.IOBase)

# Open a normal file and type check



f = open('foo.txt')
isinstance(f, IStream)     # Returns True
```

应该提到的是，@abstractmethod同样可以施加到静态方法、类方法和property属性上。只要确保以合适的顺序进行添加即可，这里@abstractmethod要紧挨着函数定义。示例如下：

```
from abc import


ABCMeta, abstractmethod

class A


(metaclass=ABCMeta):
    @property
```

```
    @abstractmethod
    def
 
 
name(self):
        pass
 
 
 
 
    @name.setter
    @abstractmethod
    def
 
 
name(self, value):
        pass
 
 
 
 
    @classmethod
    @abstractmethod
    def
 
 
method1(cls):
        pass
 
 
 
 
    @staticmethod
    @abstractmethod
    def
 
 
method2():
        pass
```

8.12.3 讨论

标准库中已经预定义好了一些抽象基类。collections模块中定义了多个和容器还有迭代器（序列、映射、集合等）相关的抽象基类。numbers库中定义了和数值对象（整数、浮点数、复数等）相关的抽象基类。io库中定义了和I/O处理相关的抽象基类。

可以使用这些预定义好的抽象基类来执行更加一般化的类型检查。下面是一些例子：

```
import collections



# Check if x is a sequence


if


isinstance(x, collections.Sequence):
    ...

# Check if x is iterable



if


isinstance(x, collections.Iterable):
```

```
    ...

# Check if x has a size



if


isinstance(x, collections.Sized):
    ...

# Check if x is a mapping



if


isinstance(x, collections.Mapping):
    ...
```

应该提到的是，在写作本节时，某些库和模块并没有像我们所期望的那样利用预定义好的抽象基类。例如：

```
from decimal import


Decimal
import numbers




x = Decimal('3.4')
isinstance(x, numbers.Real)     # Returns False
```

虽然从技术上说3.4是一个实数，由于我们无意中将浮点数和小数混在一起，这里的类型检查没有起到应有的作用。因此，如果使用了抽象基类的功能，明智的做法是仔细编写测试用例来验证其行为是否是所期待的。

尽管抽象基类使得类型检查变得更容易了，但不应该在程序中过度使用它。Python的核心在于它是一种动态语言，它带来了极大的灵活性。如果处处都强制实行类型约束，则会使得代码变得更加复杂，而这本不应该如此。我们应该拥抱Python的灵活性。

8.13 实现一种数据模型或类型系统

8.13.1 问题

我们想定义各种各样的数据结构，但是对于某些特定的属性，我们想对允许赋给它们的值强制添加一些限制。

8.13.2 解决方案

在这个问题中，基本上我们面对的任务就是在设定特定的实例属性时添加检查或者断言。为了做到这点，需要对每个属性的设定做定制化处理，因此应该使用描述符来完成。

下面的代码使用描述符实现了一个类型系统以及对值进行检查的框架：

```
# Base class. Uses a descriptor to set a value

class Descriptor

```

```
:
    def

__init__(self, name=None, **opts):
        self.name = name
        for

key, value in

opts.items():
            setattr(self, key, value)

    def

__set__(self, instance, value):
        instance.__dict__[self.name] = value

# Descriptor for enforcing types

class Typed

(Descriptor):
    expected_type = type(None)

    def

__set__(self, instance, value):
        if not

isinstance(value, self.expected_type):
            raise TypeError

('expected ' + str(self.expected_type))
        super().__set__(instance, value)
```

```
# Descriptor for enforcing values



class Unsigned


(Descriptor):
    def


__set__(self, instance, value):
        if


value < 0:
            raise ValueError


('Expected >= 0')
        super().__set__(instance, value)

class MaxSized


(Descriptor):
    def


__init__(self, name=None, **opts):
        if


'size' not in


opts:
            raise TypeError


('missing size option')
        super().__init__(name, **opts)

    def
```

```
__set__(self, instance, value):
        if


len(value) >= self.size:
            raise ValueError


('size must be < ' + str(self.size))
        super().__set__(instance, value)
```

这些类可作为构建一个数据模型或者类型系统的基础组件。让我们继续，下面这些代码实现了一些不同类型的数据：

```
class Integer


(Typed):
    expected_type = int

class UnsignedInteger


(Integer, Unsigned):
    pass




class Float


(Typed):
    expected_type = float

class UnsignedFloat
```

```
(Float, Unsigned):
    pass



class String


(Typed):
    expected_type = str

class SizedString


(String, MaxSized):
    pass
```

有了这些类型对象，现在就可以像这样定义一个类了：

```
class Stock


:
    # Specify constraints



    name = SizedString('name',size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')
    def
```

```
__init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = pric
```

有了这些约束后，就会发现现在对属性进行赋值是会进行验证的。示例如下：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 23, in __set__
    raise ValueError


('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in __set__
    raise TypeError


('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 35, in __set__
    raise ValueError
```

```
('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>
```

可以运用一些技术来简化在类中设定约束的步骤。一种方法是使用类装饰器，示例如下：

```
# Class decorator to apply constraints



def


check_attributes(**kwargs):
    def


decorate(cls):
        for


key, value in


kwargs.items():
            if


isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else



:
                setattr(cls, key, value(key))
```

```
        return


cls
    return


decorate

# Example



@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)
class Stock


:
    def


__init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

另一种方法是使用元类，示例如下：

```
# A metaclass that applies checking



class checkedmeta


(type):
    def
```

```
__new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors



        for


key, value in


methods.items():
            if


isinstance(value, Descriptor):
                value.name = key
        return


type.__new__(cls, clsname, bases, methods)
# Example



class Stock


(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()
    def


__init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

8.13.3 讨论

本节涉及了好几种高级技术，包括描述符、mixin类、对super()的使用、类装饰器以及元类。在这里涵盖所有这些主题的基础知识显然是不现实的，读者可以在其他章节中找到相关的示例（参阅8.9、8.18、9.12以及9.19节）。但是，还是有几个微妙之处值得我们讨论。

首先，在Descriptor基类中会发现有一个__set__()方法，但是却没有与之对应的__get__()方法。如果一个描述符所做的仅仅只是从底层的实例字典中提取出具有相同名称的值，那么定义__get__()就是不必要的了。实际上，在这里定义__get__()反而会让程序运行得更慢。因此，本节只会把重点放在对__set__()的实现上。

本节中各个描述符类的总体设计是基于mixin类的。例如，Unsigned和MaxSized类是用来和其他从Typed类中继承而来的描述符类混合在一起使用的。要处理某种特定的数据类型，我们使用多重继承来将所需要的功能联合在一起使用。

我们也会注意到所有描述符的__init__()方法已经被编写为具有相同的签名形式，其中涉及关键字参数**opts。MaxSized类会在opts中寻找它所需要

的属性，但是会将其传递给基类Descriptor，然后在基类中完成实际的设定。像这样的组合类（尤其是mixin），一个棘手的地方在于我们并非总是知道这些类是如何串联起来的，或者super()到底会调用些什么。基于这个原因，需要保证让所有可能出现的组合类都能正常工作。

各种类型类（type classs）的定义比如Integer、Float以及String展示了一项有用的技术，即，使用类变量来定制化实现。描述符Typed仅仅是寻找一个expected_type属性，该属性是由那些子类所提供的。

使用类装饰器或者元类常常可以简化用户代码。我们会发现在这些例子中，用户不再需要多次输入属性名了。示例如下：

```
# Normal



class Point



:
    x = Integer('x')
    y = Integer('y')

# Metaclass
```

```
class Point


(metaclass=checkedmeta):
    x = Integer()
    y = Integer()
```

实现类装饰器和元类的代码会扫描类字典，寻找描述符。当找到描述符后，它们会根据键的值自动填入描述符的名称。

在所有方法中，类装饰器可以提供最大的灵活性和稳健性。第一，这种解决方案不依赖于任何高级的机制，比如说元类。第二，装饰器可以很容易地根据需要在类定义上添加或者移除。例如，在装饰器中，可以有一个选项来简单地忽略掉添加的检查机制。这样就能让检查机制可以根据需要随意打开或关闭（调试环境对比生产环境）。

最后，采用类装饰器的解决方案也可以用来取代mixin类、多重继承以及对super()函数的使用。下面就是使用类装饰器的备选方案：

```
# Base class. Uses a descriptor to set a value


class Descriptor


```

```
:
    def


 __init__(self, name=None, **opts):
        self.name = name
        for


 key, value in


 opts.items():
            setattr(self, key, value)

    def


 __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Decorator for applying type checking



def


 Typed(expected_type, cls=None):
    if


 cls is


 None:
        return lambda


 cls: Typed(expected_type, cls)

    super_set = cls.__set__
    def
```

```
 __set__(self, instance, value):
        if


 not


 isinstance(value, expected_type):
            raise TypeError


('expected ' + str(expected_type))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return


 cls

# Decorator for unsigned values



def


 Unsigned(cls):
    super_set = cls.__set__
    def


 __set__(self, instance, value):
        if


 value < 0:
            raise ValueError


('Expected >= 0')
        super_set(self, instance, value)
    cls.__set__ = __set__
    return
```

```
 cls

# Decorator for allowing sized values



def


 MaxSized(cls):
    super_init = cls.__init__
    def


 __init__(self, name=None, **opts):
        if


 'size' not in


 opts:
            raise TypeError


('missing size option')
        super_init(self, name, **opts)
    cls.__init__ = __init__

    super_set = cls.__set__
    def


 __set__(self, instance, value):
        if


 len(value) >= self.size:
            raise ValueError


('size must be < ' + str(self.size))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return
```

```
 cls

# Specialized descriptors



@Typed(int)
class Integer


(Descriptor):
    pass




@Unsigned
class UnsignedInteger


(Integer):
    pass




@Typed(float)
class Float


(Descriptor):
    pass




@Unsigned
class UnsignedFloat


(Float):
    pass
```

```
@Typed(str)
class String


(Descriptor):
    pass




@MaxSized
class SizedString


(String):
    pass
```

在这个备选方案中定义的类能够像之前那样以完全相同的方式工作（之前的示例代码都不改变），只是每个部分都会比以前运行得更快。例如，对设定一个类型属性做简单的计时测试就能发现，采用类装饰器的方案运行速度要比采用mixin类的方案几乎快上100%。读到这里你难道还会不开心吗？

8.14 实现自定义的容器

8.14.1 问题

我们想实现一个自定义的类，用来模仿普通的内建容器类型比如列表或者字典的行为。但是，我们并不完全确定需要实现什么方法来完成。

8.14.2 解决方案

collections库中定义了各种各样的抽象基类，当实现自定义的容器类时它们会非常有用。为了说明清楚，假设我们希望自己的类能够支持迭代操作。要做到这点，只要简单地从collections.Iterable中继承即可，就像下面这样：

```
import collections



class A


(collections.Iterable):
    pass
```

从collections.Iterable中继承的好处就是可以确保必须实现所有所需的特殊方法。如果不这么做，那么在实例化时就会得到错误信息：

```
>>>


 a = A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract metho
>>>
```

要修正这个错误，只要在类中实现所需的__iter__()方法即可（参见4.2和4.7节）。

在collections库中还有其他一些值得一提的类，包括Sequence、MutableSequence、Mapping、MutableMapping、Set以及MutableSet。这些类中有许多是按照功能层次的递增来进行排列的（例如，Container、Iterable、Sized、Sequence以及MutableSequence就是一种递增式的排列）。再次说明，只要简单地对这些类进行实例化操作，就可以知道需要实现哪些方法才能让自定义的容器具有相同的行为：

```
>>> import collections



>>>



 collections.Sequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstrac
__getitem__, __len__
>>>
```

下面有个简单的例子。我们在自定义类中实现了上述所需的方法，创建了一个Sequence类，且元素总是以排序后的顺序进行存储（我们的例子实现的不是很高效，但能说明大意）：

```

import collections



import bisect





class SortedItems



(collections.Sequence):
    def



 __init__(self, initial=None):
        self._items = sorted(initial) if initial is None else []
```

```
    # Required sequence methods

    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location

    def add(self, item):
        bisect.insort(self._items, item)
```

下面是使用这个类的例子：

```
>>> items = SortedItems([5, 1, 3])
>>>
```

```
 list(items)
[1, 3, 5]
>>>

 items[0]
1
>>>

 items[-1]
5
>>>

 items.add(2)
>>>

 list(items)
[1, 2, 3, 5]
>>>

 items.add(-10)
>>>

 list(items)
[-10, 1, 2, 3, 5]
>>>

 items[1:4]
[1, 2, 3]
>>>

 3 in

 items
True
```

```
>>>


 len(items)
5
>>>


 for


n in


 items:
...     print


(n)
...
-10
1
2
3
5
>>>
```

可以看到，SortedItems的实例所表现出的行为和一个普通的序列对象完全一样，并且支持所有常见的操作，包括索引、迭代、len()、是否包含（in操作符）甚至是分片。

顺便说一句，本节中用到的bisect模块能够方便地让列表中的元素保持有序。bisect.insort()函数能够将元素插入到列表中且让列表仍然保持有序。

8.14.3　讨论

从collections库中提供的抽象基类继承，可确保我们的自定义容器实现了所有所需的方法。但是，这种继承也便于我们做类型检查。

例如，我们的自定义容器将能够满足各种各样的类型检查：

```
>>>

 items = SortedItems()
>>> import collections



>>>

 isinstance(items, collections.Iterable)
True
>>>

 isinstance(items, collections.Sequence)
True
>>>

 isinstance(items, collections.Container)
True
>>>

 isinstance(items, collections.Sized)
True
```

```
>>>

 isinstance(items, collections.Mapping)
False
>>>
```

collections模块中的许多抽象基类还针对常见的容器方法提供了默认实现。为了说明，假设有一个类从collections.MutableSequence中继承而来，就像这样：

```
class Items

(collections.MutableSequence):
    def

 __init__(self, initial=None):
        self._items = list(initial) if

 initial is

 None else

 []

    # Required sequence methods


    def
```

```
 __getitem__(self, index):
        print

('Getting:', index)
        return

 self._items[index]

    def

 __setitem__(self, index, value):
        print

('Setting:', index, value)
        self._items[index] = value

    def

 __delitem__(self, index):
        print

('Deleting:', index)
        del

 self._items[index]

    def

 insert(self, index, value):
        print

('Inserting:', index, value)
        self._items.insert(index, value)

    def
```

```
 __len__(self):
        print

('Len')
        return

 len(self._items)
```

如果创建一个Items实例，就会发现它几乎支持列表所有的核心方法（例如append()、remove()、count()等）。这些方法在实现的时候只使用了所需要的那些特殊方法。下面的交互式会话说明了这一点:

```
>>>

 a = Items([1, 2, 3])
>>>

 len(a)
Len
3
>>>

 a.append(4)
Len
Inserting: 3 4
>>>
```

```
 a.append(2)
Len
Inserting: 4 2
>>>


 a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
>>>


 a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

本节仅仅只对Python的抽象类功能做了简要的介绍。numbers模块中提供了与数值数据类型相关的类似的抽象基类。要获得更多有关如何创建自己的抽象基类的信息，请参阅8.12节。

8.15 委托属性的访问

8.15.1 问题

我们想在访问实例的属性时能够将其委托（delegate）到一个内部持有的对象上，这可以作为继承的替代方案或者是为了实现一种代理机制。

8.15.2 解决方案

简单地说，委托是一种编程模式。我们将某个特定的操作转交给（委托）另一个不同的对象实现。通常来说，最简单的委托形式看起来是这样的：

```
class A

:
    def

spam(self, x):
        pass



    def
```

```
foo(self):
        pass



class B

:
    def

__init__(self):
        self._a = A()

    def

spam(self, x):
        # Delegate to the internal self._a instance


        return

self._a.spam(x)

    def

foo(self):
        # Delegate to the internal self._a instance


        return

self._a.foo()

    def
```

```
bar(self):
        pass
```

如果仅有几个方法需要委托，编写像上面那样的代码是非常简单的。但是，如果有许多方法都需要委托，另一种实现方式是定义__getattr__()方法，就像下面这样：

```
class A

:
    def

spam(self, x):
        pass



    def

foo(self):
        pass



class B

```

```
:
    def

__init__(self):
        self._a = A()

    def

bar(self):
        pass

    # Expose all of the methods defined on class A

    def

__getattr__(self, name):
        return

getattr(self._a, name)
```

__getattr__()方法能用来查找所有的属性。如果代码中尝试访问一个并不存在的属性，就会调用这个方法。在上面的代码中，我们在访问B中未定义的方法时就能把这个操作委托给A。示例如下：

```
b = B()
b.bar()     # Calls B.bar() (exists on B)
```

```
b.spam(42)     # Calls B.__getattr__('spam') and delegates to A.s
```

委托的另一个例子就是在实现代理时。示例如下：

```
# A proxy class that wraps around another object, but

# exposes its public attributes

class Proxy

:
    def

__init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj

    def

__getattr__(self, name):
        print
```

```
('getattr:', name)
        return


getattr(self._obj, name)

    # Delegate attribute assignment



    def


__setattr__(self, name, value):
        if


name.startswith('_'):
            super().__setattr__(name, value)
        else


:
            print


('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion



    def


__delattr__(self, name):
        if


name.startswith('_'):
            super().__delattr__(name)
        else
```

```
:
            print

('delattr:', name)
            delattr(self._obj, name)
```

要使用这个代理类，只要简单地用它包装另一个实例即可。示例如下：

```
class Spam

:
    def

__init__(self, x):
        self.x = x
    def

bar(self, y):
        print

('Spam.bar:', self.x, y)

# Create an instance

s = Spam(2)

# Create a proxy around it

p = Proxy(s)
```

```
# Access the proxy

print

(p.x)     # Outputs 2


p.bar(3)     # Outputs "Spam.bar: 2 3"


p.x = 37     # Changes s.x to 37
```

通过自定义实现属性的访问方法，就可以对代理进行定制化处理，让其表现出不同的行为（例如，访问日志、只允许只读访问等）。

8.15.3 讨论

委托有时候可以作为继承的替代方案。例如，不要编写下面这样的代码：

```
class A

:
```

```
    def

spam(self, x):
        print

('A.spam', x)

    def

foo(self):
        print

('A.foo')

class B

(A):
    def

spam(self, x):
        print

('B.spam')
        super().spam(x)

    def

bar(self):
        print

('B.bar')
```

用到了委托的实现方案则会是这样：

```
class A

:
    def

spam(self, x):
        print

('A.spam', x)

    def

foo(self):
        print

('A.foo')

class B

:
    def

__init__(self):
        self._a = A()

    def

spam(self, x):
        print

('B.spam', x)
```

```
        self._a.spam(x)

    def


bar(self):
        print


('B.bar')

    def


__getattr__(self, name):
        return


getattr(self._a, name)
```

有时候当直接使用继承可能没多大意义，或者我们想更多地控制对象之间的关系（例如只暴露出特定的方法、实现接口等），此时使用委托会很有用。

当使用委托来实现代理时，这里还有几个细节需要注意。首先，__getattr__()实际上是一个回滚（fallback）方法，它只会在某个属性没有找到的时候才会调用。因此，如果访问的是代理实例本身的属性（例如本例中的_obj属性），这个方法就不会被触发调用。其次，__setattr__()和__delattr__()方法需要添加一点额外的逻辑来区分代理实例本身的属性和内部对象_obj上的属性。常用的惯例是代理

类只委托那些不以下划线开头的属性（即，代理类只暴露内部对象中的“公有”属性）。

同样需要重点强调的是__getattr__()方法通常不适用于大部分名称以双下划线开头和结尾的特殊方法。例如，考虑下面这个类：

```
class ListLike

:
    def

__init__(self):
        self._items = []
    def

__getattr__(self, name):
        return

getattr(self._items, name)
```

如果尝试创建一个ListLike对象，就会发现它能支持常见的列表方法，例如append()和insert()。但是，却无法支持len()、查找元素等操作。示例如下：

```
>>> a = ListLike()
>>> a.append(2)
```

```
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

要支持不同的操作，必须自行手动委托相应的特殊方法。示例如下：

```
class ListLike


:
    def


__init__(self):
        self._items = []
    def


__getattr__(self, name):
        return


getattr(self._items, name)

    # Added special methods to support certain list operations



    def
```

```
__len__(self):
        return

len(self._items)
    def

__getitem__(self, index):
        return

self._items[index]
    def

__setitem__(self, index, value):
        self._items[index] = value
    def

__delitem__(self, index):
        del

self._items[index]
```

请参见11.8节中的另一个例子，我们在创建代理类时利用委托来完成远端过程调用。

8.16 在类中定义多个构造函数

8.16.1 问题

我们正在编写一个类，但是想让用户能够以多种方式创建实例，而不局限于__init__()提供的这一种。

8.16.2 解决方案

要定义一个含有多个构造函数的类，应该使用类方法。下面是一个简单的示例：

```
import time



class Date

:
    # Primary constructor


    def

__init__(self, year, month, day):
        self.year = year
```

```
        self.month = month
        self.day = day

    # Alternate constructor



    @classmethod
    def


today(cls):
        t = time.localtime()
        return


cls(t.tm_year, t.tm_mon, t.tm_mday)
```

要使用这个备选的构造函数，只要把它当做函数来调用即可，例如Date.today()。示例如下：

```
a = Date(2012, 12, 21)     # Primary



b = Date.today()            # Alternate
```

8.16.3 讨论

类方法的一大主要用途就是定义其他可选的构造函数。类方法的一个关键特性就是把类作为其接

收的第一个参数（cls）。我们会注意到，类方法中会用到这个类来创建并返回最终的实例。尽管十分微不足道，但正是这一特性使得类方法能够在继承中被正确使用。示例如下：

```
class NewDate

(Date):
    pass



c = Date.today()     # Creates an instance of Date (cls=Date)


d = NewDate.today()     # Creates an instance of NewDate (cls=New
```

当定义一个有着多个构造函数的类时，应该让__init__()函数尽可能简单——除了给属性赋值之外什么都不做。如果需要的话，可以在其他备选的构造函数中选择实现更高级的操作。

与单独定义一个类方法不同的是，我们可能会倾向于让__init__()方法支持不同的调用约定。示例如下：

```
class Date
```

```
:
    def

__init__(self, *args):
        if

len(args) == 0:
            t = time.localtime()
            args = (t.tm_year, t.tm_mon, t.tm_mday)
        self.year, self.month, self.day = args
```

尽管这种技术在某些情况下是行得通的，但常常会使代码变得难以理解也不好维护。比如说，这种实现不会展示出有用的帮助字符串（没有参数名称）。此外，创建Date实例的代码也会变得不那么清晰。比较下面几种方式就能很容易看出区别：

```
a = Date(2012, 12, 21)       # Clear. A specific date.

b = Date()                  # ??? What does this do?

# Class method version

c = Date.today()         # Clear. Today's date.
```

根据上面的示例，Date.today()会调用Date.__init__()方法，以合适的年份、月份和日期为参数实例化一个Date对象。如果有必要的话，甚至可以不调用__init__()方法就创建出类实例。我们将在下一节描述这种技术。

8.17 不通过调用init来创建实例

8.17.1 问题

我们需要创建一个实例，但是出于某些原因想绕过__init__()方法，用别的方式来创建。

8.17.2 解决方案

可以直接调用类的__new__()方法来创建一个未初始化的实例。例如，考虑下面这个类：

```
class Date


:
    def


__init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

采用下面的方法可以在不调用__init__()的情况下创建一个Date实例：

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

可以看到，得到的实例是未经初始化的。因此，给实例变量设定合适的初始值现在就成了我们的责任。示例如下：

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for


key, value in


data.items():
...


    setattr(d, key, value)
...



>>> d.year
2012
>>> d.month
8
>>>
```

8.17.3 讨论

当需要以非标准的方式来创建实例时常常会遇到需要绕过__init__()的情况。比如反序列化（deserializing）数据，或者实现一个类方法将其作为备选的构造函数，都属于这种情况。例如，在前面给出的Date类中，有人可能会定义一个可选的构造函数today()：

```
from time import


localtime

class Date


:
    def


__init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def


today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return
```

```
d
```

类似地，假设正在反序列化JSON数据，要产生一个下面这样的字典：

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

如果想将这个字典转换为一个Date实例，只要使用解决方案中给出的技术即可。

当需要以非标准的方式创建实例时，通常最好不要对它们的实现做过多假设。因此，一般来说不要编写直接操纵底层实例字典__dict__的代码，除非能保证它已被定义。否则，如果类中使用了__slots__、property属性、描述符或者其他高级技术，那么代码就会崩溃。通过使用setattr()来为属性设定值，代码就会尽可能的通用。

8.18 用Mixin技术来扩展类定义

8.18.1 问题

我们有一些十分有用的方法，希望用它们来扩展其他类的功能。但是，需要添加方法的这些类之间并不一定属于继承关系。因此，没法将这些方法直接关联到一个共同的基类上。

8.18.2 解决方法

本节提到的问题在需要对类进行定制化处理时通常会出现。例如，某个库提供了一组基础类以及一些可选的定制化方法，如果用户需要的话可以自行添加。

为了说明清楚，现在假设我们有兴趣将各式各样的定制化处理方法（例如，日志记录、类型检查等）添加到映射型对象（mapping object）上。下面有一组mixin类来完成这项任务：

```
class LoggedMappingMixin


:
    '''
```

```
    Add logging to get/set/delete operations for debugging.


    '''
    __slots__ = ()

    def


 __getitem__(self, key):
        print


('Getting ' + str(key))
        return


 super().__getitem__(key)

    def


 __setitem__(self, key, value):
        print


('Setting {} = {!r}'.format(key, value))
        return


 super().__setitem__(key, value)

    def


 __delitem__(self, key):
        print


('Deleting ' + str(key))
        return


 super().__delitem__(key)
```

```
class SetOnceMappingMixin

:
    '''
    Only allow a key to be set once.


    '''
    __slots__ = ()
    def

 __setitem__(self, key, value):
        if

 key in

 self:
            raise KeyError

(str(key) + ' already set')
        return

 super().__setitem__(key, value)

class StringKeysMappingMixin

:
    '''
    Restrict keys to strings only


    '''
    __slots__ = ()
    def
```

```
 __setitem__(self, key, value):
        if


 not


 isinstance(key, str):
            raise TypeError


('keys must be strings')
        return


 super().__setitem__(key, value)
```

这些类本身是无用的。实际上，如果实例化它们中的任何一个，一点儿有用的事情都做不了（除了会产生异常之外）。相反，这些类存在的意义是要和其他映射型类通过多重继承的方式混合在一起使用。示例如下：

```
>>> class LoggedDict


(LoggedMappingMixin, dict):
...     pass




...
>>>


 d = LoggedDict()
>>>
```

```
 d['x'] = 23
Setting x = 23
>>>

 d['x']
Getting x
23
>>>

 del

 d['x']
Deleting x

>>> from collections import

 defaultdict
>>> class SetOnceDefaultDict

(SetOnceMappingMixin, defaultdict):
...     pass


...
>>>

 d = SetOnceDefaultDict(list)
>>>

 d['x'].append(2)
>>>

 d['y'].append(3)
>>>
```

```
 d['x'].append(10)
>>>


 d['x'] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 24, in __setitem__
    raise KeyError


(str(key) + ' already set')
KeyError: 'x already set'

>>> from collections import


 OrderedDict
>>> class StringOrderedDict


(StringKeysMappingMixin,
...                                 SetOnceMappingMixin,
...                                 OrderedDict):
...     pass




...
>>>


 d = StringOrderedDict()
>>>


 d['x'] = 23
>>>


 d[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "mixin.py", line 45, in __setitem__
    '''
TypeError: keys must be strings
>>>


 d['x'] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 46, in __setitem__
    __slots__ = ()
  File "mixin.py", line 24, in __setitem__
    if


 key in self:
KeyError: 'x already set'
>>>
```

在上面的示例中，可以发现这些mixin类和其他已有的类（例如：dict、defaultdict、OrderedDict）结合在了一起。当它们混合在一起时，所有的类通过一起工作提供所需的功能。

8.18.3 讨论

Python标准库中到处都是mixin类的身影，大部分都是为了扩展其他类的功能而创建的，就和我们展示的示例类似。mixin类也是多重继承的主要用途之一。例如，如果正在编写网络功能方面的代码，通常可以使用socketserver模块中的ThreadingMixIn类来为其他网络相关的类添加对线

程的支持。例如，下面是一个多线程版的XML-RPC服务器：

```
from xmlrpc.server import

 SimpleXMLRPCServer
from socketserver import

 ThreadingMixIn
class ThreadedXMLRPCServer

(ThreadingMixIn, SimpleXMLRPCServer):
    pass
```

我们在大型的库和框架中也能常看到mixin类——同样地，一般都是为了对已有的类增加一些可选的功能特性。

关于mixin类的理论，历史上有着许多讨论。但是，我们不再深入挖掘所有的细节，只需要记住几个重要的实现细节就够了。

首先，mixin类绝不是为了直接实例化而创建的。例如，本节中所有的mixin类都不能独自工作。它们必须同另一个实现了所需的映射功能的类

混合在一起用才行。同样地，socketserver模块中的ThreadingMixIn类也必须同某个合适的server类混合在一起用才行——光靠它自己没用。

其次，mixin类一般来说是没有状态的。这意味着mixin类没有__init__()方法，也没有实例变量。在本节中，我们定义的__slots__ = ()就是一种强烈的提示，这表示mixin类没有属于自己的实例数据。

如果考虑定义一个拥有__init__()方法以及实例变量的mixin类，请注意这里会有极大的风险，因为这个类并不知道自己要和哪些其他的类混合在一起。因此，任何要创建出的实例变量都必须以某种方式加以命名，以此避免出现命名冲突。此外，mixin类的__init__()方法必须要能合适地调用其他混合进来的类的__init__()方法。一般来说这很难实现，因为不知道其他类的参数签名是什么。至少，我们必须得实现非常通用的参数签名，这需要用到*arg*、* kwargs。如果mixin类的__init__()方法自身还带有参数，那么那些参数应该只能通过关键字来指定，并且在命名上还得和其他参数区分开，避免命名冲突。对于定义了一个__init__()方法且接受一个关键字参数的mixin类，下面给出了一种可能的实现方法：

```
class RestrictKeysMixin


:
    def


 __init__(self, *args, _restrict_key_type, **kwargs):
        self.__restrict_key_type = _restrict_key_type
        super().__init__(*args, **kwargs)

    def


 __setitem__(self, key, value):
        if not


 isinstance(key, self.__restrict_key_type):
            raise TypeError


('Keys must be ' + str(self.__restrict_key_type))
        super().__setitem__(key, value)
```

下面的例子展示了这个类应该如何使用：

```
>>> class RDict


(RestrictKeysMixin, dict):
...     pass




...
>>>
```

```
 d = RDict(_restrict_key_type=str)
>>>


 e = RDict([('name','Dave'), ('n',37)], _restrict_key_type=str)
>>>


 f = RDict(name='Dave', n=37, _restrict_key_type=str)
>>>


 f
{'n': 37, 'name': 'Dave'}
>>>


 f[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 83, in __setitem__
    raise TypeError


('Keys must be ' + str(self.__restrict_key_type))
TypeError: Keys must be <class 'str'>
>>>
```

在这个例子中，可以注意到初始化RDict()时仍然带有可被dict()所接受的参数，但是有一个额外的关键字参数restrict_key_type是提供给mixin类的。

最后，使用super()函数是必要的，这也是编写mixin类的关键部分。在解决方案中，这些类重新定义了一些特定的关键方法，比如__getitem__()和__setitem__()。但是，它们也需要调用这些方法的

原始版本。通过使用super()，将这个任务转交给了方法解析顺序（MRO）上的下一个类。本节中的这部分内容对于Python新手来说可能不是那么容易理解，因为我们在没有父类的类中使用了super()（初看上去感觉好像是个错误）。但是，对于类似下面这样的类定义：

```
class LoggedDict


(LoggedMappingMixin, dict):
    pass
```

在LoggedMappingMixin中使用super()函数会把任务转交到多重继承列表中的下一个类上。也就是说，在LoggedMappingMixin中调用super().__getitem__()实际上会调用dict.__getitem__()。如果没有这种行为，mixin类根本没法正常工作。

实现mixin的另一种方法是利用类装饰器。例如，考虑如下的代码：

```
def
```

```
LoggedMapping(cls):
    cls_getitem = cls.__getitem__
    cls_setitem = cls.__setitem__
    cls_delitem = cls.__delitem__

    def


__getitem__(self, key):
        print


('Getting ' + str(key))
        return


cls_getitem(self, key)

    def


__setitem__(self, key, value):
        print


('Setting {} = {!r}'.format(key, value))
        return


cls_setitem(self, key, value)

    def


__delitem__(self, key):
        print


('Deleting ' + str(key))
        return


cls_delitem(self, key)
```

```
    cls.__getitem__ = __getitem__
    cls.__setitem__ = __setitem__
    cls.__delitem__ = __delitem__
    return


 cls
```

我们把这个函数作为装饰器添加到类定义上。例如：

```
@LoggedMapping
class LoggedDict


(dict):
    pass
```

如果试着这么做，就会发现能得到相同的行为，但是却完全不再涉及多重继承了。相反，装饰器在这里只是对类定义做了一点点修改，从而替换掉特定的方法。有关类装饰器的更多细节可在9.12节中找到。

8.13节中有一个高级的示例，其中同时用到了mixin技术和类装饰器。

8.19　实现带有状态的对象或状态机

8.19.1　问题

我们想实现一个状态机，或者让对象可以在不同的状态中进行操作。但是我们并不希望代码里会因此出现大量的条件判断。

8.19.2　解决方案

在某些应用程序中，我们可能会让对象根据某种内部状态来进行不同的操作。例如，考虑下面这个代表网络连接的类：

```
class Connection

:
    def

__init__(self):
        self.state = 'CLOSED'

    def

read(self):
```

```
        if

self.state != 'OPEN':
            raise RuntimeError

('Not open')
        print

('reading')

    def

write(self, data):
        if

self.state != 'OPEN':
            raise RuntimeError

('Not open')
        print

('writing')

    def

open(self):
        if

self.state == 'OPEN':
            raise RuntimeError

('Already open')
        self.state = 'OPEN'

    def
```

```
close(self):
        if


self.state == 'CLOSED':
            raise RuntimeError


('Already closed')
        self.state = 'CLOSED'
```

这份代码为我们提出了几个难题。首先，由于代码中引入了许多针对状态的条件检查，代码变得很复杂。其次，程序的性能下降了。因为普通的操作如读（read()）和写（write()）总是要在处理前先检查状态。

一个更加优雅的方式是将每种操作状态以一个单独的类来定义，然后在Connection类中使用这些状态类。示例如下：

```
class Connection


:
    def


__init__(self):
        self.new_state(ClosedConnectionState)

    def
```

```
new_state(self, newstate):
        self._state = newstate

    # Delegate to the state class



    def


read(self):
        return


self._state.read(self)

    def


write(self, data):
        return


self._state.write(self, data)

    def


open(self):
        return


self._state.open(self)

    def


close(self):
        return


self._state.close(self)
```

```
# Connection state base class


class ConnectionState


:
    @staticmethod
    def


read(conn):
        raise NotImplementedError


()

    @staticmethod
    def


write(conn, data):
        raise NotImplementedError


()

    @staticmethod
    def


open(conn):
        raise NotImplementedError


()

    @staticmethod
    def


close(conn):
        raise NotImplementedError
```

```
()

# Implementation of different states



class ClosedConnectionState


(ConnectionState):
    @staticmethod
    def


read(conn):
        raise RuntimeError


('Not open')

    @staticmethod
    def


write(conn, data):
        raise RuntimeError


('Not open')

    @staticmethod
    def


open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def


close(conn):
        raise RuntimeError
```

```
('Already closed')

class OpenConnectionState


(ConnectionState):
    @staticmethod
    def


read(conn):
        print


('reading')

    @staticmethod
    def


write(conn, data):
        print


('writing')

    @staticmethod
    def


open(conn):
        raise RuntimeError


('Already open')

    @staticmethod
    def


close(conn):
        conn.new_state(ClosedConnectionState)
```

下面的交互式会话说明了这些类的用法：

```
>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 10, in read
    return


self._state.read(self)
  File "example.py", line 43, in read
    raise RuntimeError


('Not open')
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>
```

8.19.3 讨论

编写含有大量复杂的条件判断并和各种状态纠缠在一起的代码是难以维护和解读的。本节给出的解决方案通过将各个状态分解为单独的类来避免这

个问题。

可能看起来有些奇怪，这里每种状态都用类和静态方法来实现，在每个静态方法中都把Connection类的实例作为第一个参数。产生这种设计的原因在于我们决定在不同的状态类中不保存任何实例数据。相反，所有的实例数据应该保存在Connection实例中。将所有的状态放在一个公共的基类下，这么做的大部分原因是为了帮助组织代码，并确保适当的方法得到了实现。在基类方法中出现的NotImplementedError异常是为了确保在子类中实现了所需的方法。作为替代方案，可以考虑使用8.12节中描述过的抽象基类。

另一种实现方法是考虑去直接修改实例的__class__属性。示例如下：

```
class Connection

:
    def

__init__(self):
        self.new_state(ClosedConnection)

    def

new_state(self, newstate):
        self.__class__ = newstate
```

```
    def


read(self):
        raise NotImplementedError


()

    def


write(self, data):
        raise NotImplementedError


()

    def


open(self):
        raise NotImplementedError


()

    def


close(self):
        raise NotImplementedError


()

class ClosedConnection


(Connection):
    def


read(self):
```

```
        raise RuntimeError


('Not open')
    def


write(self, data):
        raise RuntimeError


('Not open')

    def


open(self):
        self.new_state(OpenConnection)

    def


close(self):
        raise RuntimeError


('Already closed')

class OpenConnection


(Connection):
    def


read(self):
        print


('reading')

    def


write(self, data):
```

```
        print


('writing')

    def


open(self):
        raise RuntimeError


('Already open')

    def


close(self):
        self.new_state(ClosedConnection)
```

这种实现方法的主要特点就是消除了额外的间接关系。这里不再将 Connection 和ConnectionState 作为单独的类来实现，现在我们将它们合并在一起了。随着状态的改变，实例也会修改自己的类型。示例如下：

```
>>> c = Connection()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "state.py", line 15, in read
    raise RuntimeError


('Not open')
RuntimeError: Not open
```

```
>>> c.open()
>>> c
<__main__.OpenConnection object at 0x1006718d0>
>>> c.read()
reading
>>> c.close()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>>
```

面向对象编程的拥趸不喜欢这种直接修改实例的__class__属性的做法。但是在技术上是允许这么做的。此外，这么做也会让代码的执行速度更快些，因为现在调用connection上的所有方法都不必再经过一层额外的间接步骤了。

最后，无论上面哪种技术对于实现复杂的状态机都是很有用的——尤其是在那些可能出现大量的if-elif-else块的代码中。示例如下：

```
# Original implementation



class State



:
    def



__init__(self):
        self.state = 'A'
    def
```

```
action(self, x):
        if

state == 'A':
            # Action for A

            ...
            state = 'B'
        elif

state == 'B':
            # Action for B

            ...
            state = 'C'
        elif

state == 'C':
            # Action for C

            ...
            state = 'A'

# Alternative implementation

class State

:
    def
```

```
__init__(self):
        self.new_state(State_A)

    def


new_state(self, state):
        self.__class__ = state

    def


action(self, x):
        raise NotImplementedError


()

class State_A


(State):
    def


action(self, x):
        # Action for A



        ...
        self.new_state(State_B)

class State_B


(State):
    def


action(self, x):
        # Action for B
```

```
        ...
        self.new_state(State_C)

class State_C


(State):
    def


action(self, x):
        # Action for C



        ...
        self.new_state(State_A)
```

本节大体上是基于*Design Patterns: Elements of Resuable Object-Oriented Software*（Addison-Wesley, 1995）一书中有关状态模式的内容来编写的。

8.20　调用对象上的方法，方法名以字符串形式给出

8.20.1　问题

我们想调用对象上的某个方法，现在这个方法名保存在字符串中，我们想通过它来调用该方法。

8.20.2　解决方案

对于简单的情况，可能会使用getattr()，示例如下：

```
import math



class Point


:
    def


__init__(self, x, y):
        self.x = x
        self.y = y

    def
```

```
__repr__(self):
        return

'Point({!r:},{!r:})'.format(self.x, self.y)

    def

distance(self, x, y):
        return

math.hypot(self.x - x, self.y - y)

p = Point(2, 3)
d = getattr(p, 'distance')(0, 0)     # Calls p.distance(0, 0)
```

另一种方法是使用operator.methodcaller()。示例如下：

```
import operator

operator.methodcaller('distance', 0, 0)(p)
```

如果想通过名称来查询方法并提供同样的参数反复调用该方法，那么operator.methodcall()是很有

用的。例如，如果你要对一整列点对象排序：

```
points = [
    Point(1, 2),
    Point(3, 0),
    Point(10, -3),
    Point(-5, -7),
    Point(-1, 8),
    Point(3, 2)
]

# Sort by distance from origin (0, 0)



points.sort(key=operator.methodcaller('distance', 0, 0))
```

8.20.3　讨论

调用一个方法实际上涉及两个单独的步骤，一是查询属性，二是函数调用。因此，要调用一个方法，可以使用getattr()来查询相应的属性。要调用查询到的方法，只要把查询的结果当做函数即可。

operator.methodcall()创建了一个可调用对象，而且把所需的参数提供给了被调用的方法。我们所要做的就是提供恰当的self参数即可。示例如下：

```
>>> p = Point(3, 4)
>>> d = operator.methodcaller('distance', 0, 0)
>>> d(p)
5.0
```

```
>>>
```

通过包含在字符串中的名称来调用方法，这种方式时常出现在需要模拟case语句或者访问者模式的变体中。下一节中将有更加高级的示例。

8.21　实现访问者模式

8.21.1　问题

我们需要编写代码来处理或遍历一个由许多不同类型的对象组成的复杂数据结构，每种类型的对象处理的方式都不相同。例如遍历一个树结构，根据遇到的树节点的类型来执行不同的操作。

8.21.2　解决方案

本节提到的这个问题常常出现在由大量不同类型的对象组成的数据结构的程序中。为了说明，假设我们正在编写一个表示数学运算的程序。要实现这个功能，程序中会用到一些类，示例如下：

```
class Node

:
    pass



class UnaryOperator

(Node):
```

```
    def

__init__(self, operand):
        self.operand = operand

class BinaryOperator

(Node):
    def

__init__(self, left, right):
        self.left = left
        self.right = right

class Add

(BinaryOperator):
    pass



class Sub

(BinaryOperator):
    pass



class Mul

(BinaryOperator):
    pass



class Div
```

```
(BinaryOperator):
    pass



class Negate


(UnaryOperator):
    pass



class Number


(Node):
    def


__init__(self, value):
        self.value = value
```

之后，我们可以用这些类来构建嵌套式的数据结构，就像这样：

```
# Representation of 1 + 2 * (3 - 4) / 5



t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)
```

问题不在创建这些数据结构上，而是在稍后编写处理它们的代码时。例如，给定一个表达式，程序可能要做很多事情，比如产生输出、生成指令、执行字节码到机器码的翻译等。

为了能让处理过程变得通用，一种常见的解决方案就是实现所谓的“访问者模式”。我们需要使用类似下面这样的类：

```
class NodeVisitor


:
    def


visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if


meth is


None:
            meth = self.generic_visit
        return


meth(node)

    def
```

```
generic_visit(self, node):
        raise RuntimeError


('No {} method'.format('visit_' + type(node).__name__))
```

要使用这个类，程序员从该类中继承并实现各种visit_Name()方法，这里的Name应该由节点的类型来替换。例如，如果想对表达式求值，那么可以编写这样的代码：

```
class Evaluator


(NodeVisitor):
    def


visit_Number(self, node):
        return


node.value

    def


visit_Add(self, node):
        return


self.visit(node.left) + self.visit(node.right)

    def


visit_Sub(self, node):
```

```
        return


self.visit(node.left) - self.visit(node.right)

    def


visit_Mul(self, node):
        return


self.visit(node.left) * self.visit(node.right)

    def


visit_Div(self, node):
        return


self.visit(node.left) / self.visit(node.right)

    def


visit_Negate(self, node):
        return


-node.operand
```

下面这个例子展示如何使用这个类来计算前面生成的表达式：

```
>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>
```

作为另一个完全不同的例子，下面这个类可以将表达式翻译为堆栈机（stack machine）上的指令序列：

```
class StackCode


(NodeVisitor):
    def


generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return


self.instructions

    def


visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

    def


binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def


visit_Add(self, node):
        self.binop(node, 'ADD')
```

```
    def


visit_Sub(self, node):
        self.binop(node, 'SUB')

    def


visit_Mul(self, node):
        self.binop(node, 'MUL')

    def


visit_Div(self, node):
        self.binop(node, 'DIV')

    def


unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def


visit_Negate(self, node):
        self.unaryop(node, 'NEG')
```

如何使用这个类呢？示例如下：

```
>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
>>>
```

8.21.3 讨论

本节涵盖了两个核心思想。首先是设计策略，即把操作复杂数据结构的代码和数据结构本身进行解耦。也就是说，本节中没有任何一个Node类的实现有对数据进行操作。相反，所有对数据的处理都放在特定的NodeVisitor类中实现。这种隔离使得代码变得非常通用。

本节的第二个核心思想在于对访问者类本身的实现。在访问者中，你想根据某些值比如节点类型来调度不同的处理方法。一种幼稚的做法是会编写大量的if语句，就像下面这样：

```
class NodeVisitor

:
    def

visit(self, node):
        nodetype = type(node).__name__
        if

nodetype == 'Number':
            return

self.visit_Number(node)
        elif
```

```
nodetype == 'Add':
            return

self.visit_Add(node)
        elif

nodetype == 'Sub':
            return

self.visit_Sub(node)
        ...
```

但是，很快就会发现这种做法明显行不通。除了非常繁琐之外，运行速度也很慢。如果想添加或修改要处理的节点类型则会难以维护。相反，如果通过一些小技巧将方法名构建出来，再利用getattr()函数来获取方法则会好得多。解决方案中的generic_visit()不应该匹配到任何处理方法，它是一种异常回退机制。在本节中，generic_visit()会抛出一个异常来警告程序员遇到了一个未知的节点类型。

在每个访问者类中，常常会通过对visit()方法进行递归调用来完成计算。示例如下：

```
class Evaluator
```

```
(NodeVisitor):
    ...
    def


visit_Add(self, node):
        return


self.visit(node.left) + self.visit(node.right)
```

正是由于递归才使得访问者类可以遍历整个数据结构。本质上说就是不断调用visit()直到到达某个终止节点，比如示例中的Number。递归和其他操作的确切顺序完全取决于应用程序。

应该提到的是，这种调度方法的技术在其他语言中也常用来模拟开关行为或者条件语句。例如，如果我们正在编写一个HTTP框架，我们在类中也会实现类似的方法调度：

```
class HTTPHandler


:
    def


handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)

    def
```

```
do_GET(self, request):
        ...
    def


do_POST(self, request):
        ...
    def


do_HEAD(self, request):
        ...
```

访问者模式的一个缺点就是需要重度依赖于递归。如果要处理一个深度嵌套的数据结构，那么有可能会达到Python的递归深度限制（查看sys.getrecursionlimit()的结果）。要避免这个问题，可以在构建数据结构时做一些特定的选择。例如，可以使用普通的Python 列表来替代链表，或者在每个节点中聚合更多数据，使得数据变得扁平化而不是深度嵌套。

也可以尝试利用生成器和迭代器实现非递归式的遍历算法，具体内容可参见8.22节。

在有关解析和编译的程序中使用访问者模式是非常常见的。在Python自带的ast模块中可以找到一个实现。除了可以遍历树结构之外，在遍历的同时还允许对数据结构进行改写或转换（例如添加节点

或移除节点）。具体细节可查看ast模块的源码。9.24节中展示了一个利用ast模块来处理Python源代码的例子。

8.22　实现非递归的访问者模式

8.22.1　问题

我们使用访问者模式来遍历一个深度嵌套的树结构，但由于超出了Python的递归限制而崩溃。我们想要去掉递归，但依旧保持访问者模式的编程风格。

8.22.2　解决方案

巧妙利用生成器有时候可用来消除树的遍历或查找算法中的递归。在8.21节中，我们已经给出了一个访问者类。下面是这个类的另一种实现方式，通过堆栈和生成器来驱动计算，完全不使用递归。

```
import types



class Node

:
    pass


```

```
import types


class NodeVisitor


:
    def


visit(self, node):
        stack = [ node ]
        last_result = None
        while


stack:
            try


:
                last = stack[-1]
                if


isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif


isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else


:
                    last_result = stack.pop()
            except StopIteration


:
                stack.pop()
```

```
        return

last_result

    def

_visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if

meth is

None:
            meth = self.generic_visit
        return

meth(node)

    def

generic_visit(self, node):
        raise RuntimeError

('No {} method'.format('visit_' + type(node).__name__))
```

如果使用这个类，就会发现配合之前已有的代码（可能使用了递归），程序仍然可以正常工作。实际上，我们可以用其替换上一节中的访问者类实现。例如，考虑下面的代码，其中涉及表达式树：

```
class UnaryOperator


(Node):
    def


__init__(self, operand):
        self.operand = operand

class BinaryOperator


(Node):
    def


__init__(self, left, right):
        self.left = left
        self.right = right

class Add


(BinaryOperator):
    pass




class Sub


(BinaryOperator):
    pass




class Mul


(BinaryOperator):
    pass
```

```
class Div

(BinaryOperator):
    pass



class Negate

(UnaryOperator):
    pass



class Number

(Node):
    def

__init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions



class Evaluator

(NodeVisitor):
    def

visit_Number(self, node):
        return
```

```
node.value

    def


visit_Add(self, node):
        return


self.visit(node.left) + self.visit(node.right)

    def


visit_Sub(self, node):
        return


self.visit(node.left) - self.visit(node.right)

    def


visit_Mul(self, node):
        return


self.visit(node.left) * self.visit(node.right)

    def


visit_Div(self, node):
        return


self.visit(node.left) / self.visit(node.right)

    def


visit_Negate(self, node):
        return
```

```
-self.visit(node.operand)

if


__name__ == '__main__':
    # 1 + 2*(3-4) / 5



    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)

    # Evaluate it



    e = Evaluator()
    print


(e.visit(t4))     # Outputs 0.6
```

上述代码在处理简单的表达式时是没有问题的。但是，Evaluator的实现中使用了递归，如果嵌套层次太深的话程序就会崩溃。示例如下：

```
>>> a = Number(0)
>>> for


n in
```

```
range(1, 100000):
...
     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
  File "visitor.py", line 29, in _visit
    return
meth(node)
  File "visitor.py", line 67, in visit_Add
    return
self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>
```

现在，我们把Evaluator类稍微修改一下：

```
class Evaluator
(NodeVisitor):
    def
```

```
visit_Number(self, node):
        return

node.value

    def

visit_Add(self, node):
        yield

(yield

node.left) + (yield

node.right)

    def

visit_Sub(self, node):
        yield

(yield

node.left) - (yield

node.right)

    def

visit_Mul(self, node):
        yield

(yield
```

```
node.left) * (yield

node.right)
    def

visit_Div(self, node):
        yield

(yield

node.left) / (yield

node.right)
    def

visit_Negate(self, node):
        yield

-(yield

node.operand)
```

如果再次尝试同样的试验，会发现程序突然就可以正常工作了，真是神奇！

```
>>> a = Number(0)
>>> for
```

```
n in


range(1,100000):
...


     a = Add(a, Number(n))
...



>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>
```

如果想在任意一个方法中添加自定义的处理，程序依然可以正常工作。示例如下：

```
class Evaluator


(NodeVisitor):
    ...
    def


visit_Add(self, node):
        print


('Add:', node)
        lhs = yield


node.left
        print
```

```
('left=', lhs)
        rhs = yield


node.right
        print


('right=', rhs)
        yield


lhs + rhs
    ...
```

下面是示例输出：

```
>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>
```

8.22.3　讨论

本节很好地展示了如何利用生成器和协程来控制程序的执行流。这种令人费解的技巧常常能带来很大的优势。要理解本节的内容，需要深入了解几个要点。

首先，在有关遍历树结构的问题中，为了避免使用递归，常见的策略就是利用栈或者队列来实现算法。例如，深度优先遍历完全可以实现为将第一个遇到的节点压入栈中，一旦处理结束再将其弹出。解决方案中给出的visit()方法的核心就是按照这个思路实现的。算法一开始会将初始节点压入stack列表中（这里的栈以Python列表的形式来实现），然后继续运行直到栈为空为止。在执行算法的时候，栈会根据树结构的深度进行增长。

第二个要点在于生成器中yield语句的行为。当遇到yield语句时，生成器会产生出一个值然后暂停执行。本节正是利用这个特性来取代递归。例如，现在我们不用像这样编写递归式的表达式了：

```
value = self.visit(node.left)
```

我们用下面这条语句来替代：

```
value = yield

node.left
```

在幕后，这条语句会将node.left节点发送回给visit()方法。之后，visit()就可以为该节点调用合适

的visit_Name()方法了。从某种意义上说，这几乎和递归算法恰好相反。也就是说，现在不是通过递归调用visit()来遍历树节点了，而是在处理的过程中用yield语句来暂停计算。因此，yield本质上可当做一种信号来告诉算法当前处在yield状态的节点需要先被处理，之后剩下的处理才可以继续进行。

本节中最后一个需要考虑的问题是如何传递结果。当我们使用生成器函数时，我们不能再使用return语句来发送结果了（这么做会产生SyntaxError异常）。因此，yield语句必须来承担这个责任。在本节中，如果由yield语句产生出的值是非节点类型（non-Node type）的，则认为该值是要发送给计算过程中的下一个步骤的。这正是在代码中使用变量last_return的目的所在。一般来说，last_return将保存某个访问方法上一次产生出的值。这个值会作为yield语句的返回值发送到上一个执行的方法中。例如，在下面的代码中：

```
value = yield


node.left
```

变量value将获得last_return的值，而这个值正是在为节点node.left调用访问方法时返回的结果。

以上所有要点都可以在下面的代码片段中找到：

```
try


:
    last = stack[-1]
    if


isinstance(last, types.GeneratorType):
        stack.append(last.send(last_result))
        last_result = None
    elif


isinstance(last, Node):
        stack.append(self._visit(stack.pop()))
    else



:
        last_result = stack.pop()
    except StopIteration


:
        stack.pop()
```

这段代码简单地查看栈顶并决定下一步该做什么。如果是生成器，那么就调用它的send()方法将上次得到的结果（如果有结果的话）添加到栈中以待后续处理。send()返回的值和传给yield语句的值是相同的。因此，在yield node.left这样的语句中，

send()返回的就是Node的实例node.left，并会将其放置在栈的顶部。

如果栈顶是一个Node实例，那么该实例会被替换为在该节点上调用合适的访问方法所得到的结果。正是因为这样，我们完全避免了对递归的使用。之前我们是在各个访问方法中以递归的方式直接调用visit()（参见上一节解决方案中的实现），现在不必这么做了。只要在各个访问方法中使用yield，那么程序就能正常工作。

最后，如果栈顶元素为其他值，则可认为这是某种类型的返回值。我们将其从栈中弹出然后保存到last_result中。如果栈中的下一个元素是生成器，那么就将它作为yield语句的返回值发送出去。应该提到的是，visit()的最后一个返回值也会赋给last_result。这样就使得本节中的代码也能适用于传统的递归实现。如果没有用到生成器，last_result就保存着代码中return语句的返回值。

本节中一个潜在的危险在于产生Node和非Node值之间的区别。在我们的实现中会自动遍历所有的Node实例。这意味着我们不能把Node当做返回值来进行传递。在实践中，这也许无关紧要。但是如果确实有这个需求，就需要对算法做轻微的调整。例如，可以通过引入另一个类来解决：

```
class Visit


:
    def


__init__(self, node):
        self.node = node

class NodeVisitor


:
    def


visit(self, node):
        stack = [ Visit(node) ]
        last_result = None
        while


stack:
            try


:
                last = stack[-1]
                if


isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif


isinstance(last, Visit):
                    stack.append(self._visit(stack.pop().node))
                else


:
```

```
                    last_result = stack.pop()
            except StopIteration

:
                stack.pop()
        return

last_result

    def

_visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if

meth is

None:
            meth = self.generic_visit
        return

meth(node)

    def

generic_visit(self, node):
        raise RuntimeError

('No {} method'.format('visit_' + type(node).__name__))
```

根据上面的实现，现在访问方法看起来就是这样的了：

```
class Evaluator


(NodeVisitor):
    ...
    def


visit_Add(self, node):
        yield


(yield


Visit(node.left)) + (yield


Visit(node.right))

    def


visit_Sub(self, node):
        yield


(yield


Visit(node.left)) - (yield


Visit(node.right))
    ...
```

看过本节之后，你可能会倾向于去实现一种不涉及yield的解决方案。但是，这么做会使得我们必须在代码中处理本节中已经提到过的诸多问题。例

如，要消除对递归的使用，需要维护一个栈。也需要有某种方法来管理对树结构的遍历以及调用各种访问者方法的逻辑。没有生成器的帮助，这种代码将演变成一锅大杂烩，其中混杂着对栈的操作、回调函数以及其他的组件。坦白说，使用yield的主要优势在于我们能够以优雅的风格编写出非递归式的代码，而且看起来和递归式的实现几乎一样。

8.23　在环状数据结构中管理内存

8.23.1　问题

我们的程序中创建了环状的数据结构（例如树、图、观察者模式等），但是在内存管理上却遇到了麻烦。

8.23.2　解决方案

环状数据结构的一个简单例子就是树了，这里父节点指向它的孩子，而孩子节点又会指回它们的父节点。对于像这样的代码，我们应该考虑让其中一条连接使用weakref库中提供的弱引用机制。示例如下：

```
import weakref



class Node

:
    def


__init__(self, value):
```

```
        self.value = value
        self._parent = None
        self.children = []

    def


__repr__(self):
        return


'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference



    @property
    def


parent(self):
        return


self._parent if


self._parent is


None else


self._parent()

    @parent.setter
    def


parent(self, node):
        self._parent = weakref.ref(node)

    def
```

```
add_child(self, child):
        self.children.append(child)
        child.parent = self
```

这种实现可以让父节点安静地被回收。示例如下：

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print


(c1.parent)
Node('parent')
>>> del


root
>>> print


(c1.parent)
None
>>>
```

8.23.3 讨论

环状数据结构是Python中一个多少需要一些技巧才能处理好的方面，需要仔细学习。因为普通的垃圾收集规则并不适用于环状数据结构。例如，考

虑下面的代码：

```
# Class just to illustrate when deletion occurs


class Data


:
    def


__del__(self):
        print


('Data.__del__')

# Node class involving a cycle


class Node


:
    def


__init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def


add_child(self, child):
        self.children.append(child)
        child.parent = self
```

现在，试用上面的代码，做些试验来看看有关垃圾收集中的一些微妙问题：

```
>>> a = Data()
>>> del


a             # Immediately deleted




Data.__del__
>>> a = Node()
>>> del


a             # Immediately deleted




Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del


a             # Not deleted (no message)




>>>
```

可以看到，除了最后那种涉及成环的情况，其他的对象都可以立刻得到删除。原因在于Python的垃圾收集器是基于简单的引用计数规则来实现的。

当对象的引用计数为0时就会被立刻删除掉。而对于环状数据结构来说这绝不可能发生。因为在最后那种情况中，由于父节点和子节点互相引用对方，引用计数不会为0。

要处理环状数据结构，还有一个单独的垃圾收集器会定期运行。但是，一般来说我们不知道它会在何时运行。因此，没法知道环状数据结构具体会在何时被回收。如果有必要的话，可以强制运行垃圾收集器，但这么做相比于全自动的垃圾收集会有一些笨拙。

```
>>> import gc



>>> gc.collect()     # Force collection



Data.__del__
Data.__del__
>>>
```

如果环中的对象实现了自己的__del__方法的话，则情况会更糟。例如，假设有下面这样的代码：

```
# Class just to illustrate when deletion occurs
```

```
class Data


:
    def


__del__(self):
        print


('Data.__del__')

# Node class involving a cycle



class Node


:
    def


__init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    # NEVER DEFINE LIKE THIS.



    # Only here to illustrate pathological behavior



    def


__del__(self):
        del
```

```
self.data
        del


.parent
        del


.children

    def


add_child(self, child):
        self.children.append(child)
        child.parent = self
```

在这种情况下，数据结构对象永远不会被垃圾收集，我们的程序会因此而出现内存泄露！如果动手尝试一下，会发现Data.__del__消息完全没有被打印出来——即使是强制执行垃圾收集也不会：

```
>>> a = Node()
>>> a.add_child(Node()
>>> del


a             # No message (not collected)



>>> import gc



>>> gc.collect()     # No message (not collected)
```

```
>>>
```

弱引用通过消除循环引用来解决这个问题。本质上说，弱引用就是一个指向对象的指针，但不会增加对象本身的引用计数。可以通过weakref库来创建弱引用。示例如下：

```
>>> import weakref



>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
>>>
```

要提领（dereference）一个弱引用，可以像函数一样来调用它。如果提领后得到的对象还依然存在，那么就返回对象，否则就返回None。由于原始对象的引用计数并没有增加，因此可以按照普通的方式来删除它。示例如下：

```
>>> print


(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del
```

```
a
Data.__del__
>>> print


(a_ref())
None
>>>
```

通过使用弱引用，就会发现因为循环引用而出现的问题都不存在了。一旦某个对象不再被使用了，会立刻执行垃圾收集处理。请参阅8.25节中另一个有关弱引用的示例。

8.24　让类支持比较操作

8.24.1　问题

我们想使用标准的比较操作符（如>=、!=、<=等）在类实例之间进行比较，但是又不想编写大量的特殊方法。

8.24.2　解决方案

通过为每种比较操作符实现一个特殊方法，Python中的类可以支持比较操作。例如，要支持>=操作符，可以在类中定义一个__ge__()方法。虽然只定义一个方法不算什么，但如果要实现每种可能的比较操作，那么实现这么多特殊方法则很快会变得繁琐。

functools.total_ordering装饰器可用来简化这个过程。要使用它，可以用它来装饰一个类，然后定义__eq__()以及另一个比较方法（__lt__、__le__、__gt__或者__ge__）。那么装饰器就会自动为我们实现其他的比较方法。

作为示例，让我们来构建一些房子并为其添加

一些房间吧，然后根据房子的大小来进行比较：

```
from functools import


total_ordering
class Room


:
    def


__init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House


:
    def


__init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()

    @property
    def


living_space_footage(self):
        return


sum(r.square_feet for
```

```
r in

self.rooms)
    def

add_room(self, room):
        self.rooms.append(room)
    def

__str__(self):
        return

'{}: {} square foot {}'.format(self.name,
                                             self.living_space_f
                                             self.style)
    def

__eq__(self, other):
        return

self.living_space_footage == other.living_space_footage
    def

__lt__(self, other):
        return

self.living_space_footage < other.living_space_footage
```

这里，House类已经用@total_ordering来进行装

饰了。我们定义了__eq__()和__lt__()来根据房间的总面积对房子进行比较。只需要定义这两个特殊方法就能让其他所有的比较操作正常工作。示例如下：

```
# Build a few houses, and add rooms to them



h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))

h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))

h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]

print


('Is h1 bigger than h2?', h1 > h2) # prints True



print


('Is h2 smaller than h3?', h2 < h3) # prints True
```

```
print
('Is h2 greater than or equal to h1?', h2 >= h1) # Prints False

print
('Which one is biggest?', max(houses)) # Prints 'h3: 1101-square-

print
('Which is smallest?', min(houses)) # Prints 'h2: 846-square-foot
```

8.24.3　讨论

如果我们曾经编写过代码让类支持所有的基本比较操作符，那么装饰器total_ordering对我们而言就并非那么神奇：它从字面上定义了从每个比较方法到其他所有需要该方法的映射关系。因此，如果在类中定义了__lt__()，那么就会利用它来构建其他所有的比较操作符。实际上就是在类中填充以下方法：

```
class House

:
    def

__eq__(self, other):
        ...
    def

__lt__(self, other):
        ...
    # Methods created by @total_ordering


    __le__ = lambda

self, other: self < other or

self == other
    __gt__ = lambda

self, other: not

(self < other or

self == other)
    __ge__ = lambda

self, other: not

(self < other)
    __ne__ = lambda
```

```
self, other: not

self == other
```

的确，自行编写这些方法并不难，但@total_ordering让这一过程变得更加简单了。

8.25 创建缓存实例

8.25.1 问题

当创建类实例时我们想返回一个缓存引用，让其指向上一个用同样参数（如果有的话）创建出的类实例。

8.25.2 解决方案

本节提到的这个问题常常出现在当我们想确保针对某一组输入参数只会有一个类实例存在时。现实中的例子包括一些库的行为，比如在logging模块中，给定的一个名称只会关联到一个单独的logger实例。示例如下：

```
>>> import logging



>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is


b
False
>>> c = logging.getLogger('foo')
>>> a is
```

```
c
True
>>>
```

要实现这一行为，应该使用一个与类本身相分离的工厂函数。示例如下：

```
# The class in question

class Spam

:
    def

__init__(self, name):
        self.name = name

# Caching support

import weakref

_spam_cache = weakref.WeakValueDictionary()

def

get_spam(name):
    if
```

```
name not in


_spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else


:
        s = _spam_cache[name]
    return


s
```

如果你用上述实现，会发现Spam类的行为和之前展示的效果一样：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is


b
False
>>> c = get_spam('foo')
>>> a is


c
True
>>>
```

8.25.3 讨论

要想修改实例创建的规则，编写一个特殊的工厂函数常常是一种简单的方法。此时，一个常被提到的问题就是是否可以用更加优雅的方式来完成呢？

例如，我们可能会考虑重新定义类的__new__()方法：

```
# Note: This code doesn't quite work

import weakref

class Spam

:
    _spam_cache = weakref.WeakValueDictionary()
    def

__new__(cls, name):
        if

name in

cls._spam_cache:
            return

cls._spam_cache[name]
        else
```

```
:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return

self

    def

__init__(self, name):
        print

('Initializing Spam')
        self.name = name
```

初看上去，上面的代码似乎可以完成任务。但是，主要的问题在于__init__()方法总是会得到调用，无论对象实例有无得到缓存都是如此。示例如下：

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is

t
True
>>>
```

这种行为很可能不是我们想要的。因此，要解决实例缓存后会重复初始化的问题，需要采用一个稍有些不同的方法。

本节中对弱引用的运用与垃圾收集有着极为重要的关系。当维护实例缓存时，只要在程序中实际用到了它们，那么通常希望将对象保存在缓存中。WeakValueDictionary会保存着那些被引用的对象，只要它们存在于程序中的某处即可。否则，当实例不再被使用时，字典的键就会消失。示例如下：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del


a
>>> del


c
>>> list(_spam_cache)
['bar']
>>> del


b
>>> list(_spam_cache)
[]
>>>
```

对于许多程序而言，使用本节中给出的框架代码通常就足够了。但是，还可以考虑一些更加高级的实现技术。

我们立刻能想到的是，本节中的解决方案需要依赖全局变量以及一个与原始的类定义相分离的工厂函数。一种改进方式是将缓存代码放到另一个单独的管理类中，然后将这些组件粘合在一起：

```
import weakref

class CachedSpamManager

:
    def

__init__(self):
        self._cache = weakref.WeakValueDictionary()
    def

get_spam(self, name):
        if

name not in

self._cache:
            s = Spam(name)
            self._cache[name] = s
        else
```

```
:
            s = self._cache[name]
        return


s

    def


clear(self):
        self._cache.clear()

class Spam


:
    manager = CachedSpamManager()
    def


__init__(self, name):
        self.name = name

    def


get_spam(name):
        return


Spam.manager.get_spam(name)
```

这种方法的特点就是为潜在的灵活性提供了更多支持。例如，我们可以实现不同类型的缓存管理机制（以单独的类来实现），然后附加到Spam类中替换掉默认的缓存实现。其他的代码（比如

get_spam）不需要修改就能正常工作。

另一种设计上的考虑是到底要不要将类的定义暴露给用户。如果什么都不做的话，用户可以很容易创建出实例，从而绕过缓存机制：

```
>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is


b
False
>>>
```

如果预防出现这种行为对程序而言很重要，我们可以采取特定的步骤来避免。例如，可以在类名前加一个下划线，例如_Spam，这样至少可以提醒用户不应该直接去访问它。

或者，如果想为用户提供更强的提示，暗示他们不应该直接实例化Spam对象，可以让__init__()方法抛出一个异常，然后用一个类方法来实现构造函数的功能，就像下面这样：

```
class Spam


:
    def
```

```
 __init__(self, *args, **kwargs):
        raise RuntimeError


("Can't instantiate directly")

    # Alternate constructor



    @classmethod
    def


 _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
```

要使用上述代码，可以将实现缓存机制的代码修改为使用Spam._new()来创建实例，而不是使用通常所见的Spam()。示例如下：

```
import weakref




class CachedSpamManager


:
    def


 __init__(self):
        self._cache = weakref.WeakValueDictionary()
```

```
    def


 get_spam(self, name):
        if name not in


 self._cache:
            s = Spam._new(name)         # Modified creation
            self._cache[name] = s
        else



:
            s = self._cache[name]
        return


 s
```

尽管还有更加极端的方法来隐藏Spam类的可见性，但也许最好不要把问题想的过于复杂。在类名前添加下划线或者用类方法作为构造函数通常就足以给程序员带来提示了。

通过使用元类，缓存机制以及其他的创建模式（creational pattern）通常能够以更加优雅的方式得以解决。关于元类，请参阅9.13节。

---

[1] 即满足obj == eval(repr(obj))。——译者注

[2] 即，把类中定义的函数当做一种属性来使

用。——译者注

[3]　实际上是以Python元组来表示的，因为__mro__属性是只读的。——译者注

[4]　super在英语中就表示“超级的”，“极好的”，作者在这里是双关，借用函数名super来表示它的强大。——译者注

第9章　元编程

软件开发中最重要的一条真理就是“不要重复自己的工作（Don’t repeat yourself）”。也就是说，任何时候当需要创建高度重复的代码（或者需要复制粘贴源代码）时，通常都需要寻找一个更加优雅的解决方案。在Python中，这类问题常常会归类为“元编程”。简而言之，元编程的主要目标是创建函数和类，并用它们来操纵代码（比如说修改、生成或者包装已有的代码）。Python中基于这个目的的主要特性包括装饰器、类装饰器以及元类。但是，还有许多其他有用的主题——包括对象签名、用exec()来执行代码以及检查函数和类的内部结构——也进入了我们的视野。本章的主要目的是探讨各种元编程技术，通过示例来讲解如何利用这些技术来自定义Python的行为，使其能满足我们不同寻常的需求。

9.1 给函数添加一个包装

9.1.1 问题

我们想给函数加上一个包装层（wrapper layer）以添加额外的处理（例如，记录日志、计时统计）。

9.1.2 解决方案

如果需要用额外的代码对函数做包装，可以定义一个装饰器函数。示例如下：

```
import time

from functools import

wraps
def

timethis(func):
    '''

    Decorator that reports the execution time.
```

```
    '''

    @wraps(func)
    def

wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print

(func.__name__, end-start)
        return

result
    return

wrapper
```

下面是使用这个装饰器的示例：

```
>>> @timethis
... def

countdown(n):
...

     '''
```

```
...
    Counts down

...
    '''

...
    while
n > 0:
...
            n -= 1
...

>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>
```

9.1.3 讨论

装饰器就是一个函数，它可以接受一个函数作

为输入并返回一个新的函数作为输出。当像这样编写代码时：

```
@timethis
def


countdown(n):
    ...
```

和单独执行下列步骤的效果是一样的：

```
def


countdown(n):
    ...
countdown = timethis(countdown)
```

顺便插一句，内建的装饰器比如@staticmethod、@classmethod以及@property的工作方式也是一样的。比如说，下面这两个代码片段的效果是相同的：

```
class A


:
    @classmethod
    def
```

```
method(cls):
        pass



class B

:
    # Equivalent definition of a class method


    def

method(cls):
    pass


method = classmethod(method)
```

装饰器内部的代码一般会涉及创建一个新的函数，利用*args和**kwargs来接受任意的参数。本节示例中的wrapper()函数正是这么做的。在这个函数内部，我们需要调用原来的输入函数（即被包装的那个函数，它是装饰器的输入参数）并返回它的结果。但是，也可以添加任何想要添加的额外代码（例如计时处理）。这个新创建的wrapper函数会作为装饰器的结果返回，取代了原来的函数。

需要重点强调的是，装饰器一般来说不会修改调用签名，也不会修改被包装函数返回的结果。这里对*args和**kwargs的使用是为了确保可以接受任何形式的输入参数。装饰器的返回值几乎总是同调用func(*args, **kwargs)的结果一致，这里的func就是那个未被包装过的原始函数。

当初次学习装饰器时，通过一些简单的例子来入门是很容易的，就像本节给出的那个计时的例子一样。但是，如果打算在生产环境中编写装饰器，那么这里还有一些细节需要考虑。比方说，我们的解决方案中对装饰器@wraps(func)的使用就是一个容易忘记但是却很重要的技术，它可以用来保存函数的元数据。这方面的内容将在下一节中描述。如果我们要编写自己的装饰器函数，那么接下来的几节将会补充一些很重要的细节。

9.2　编写装饰器时如何保存函数的元数据

9.2.1　问题

　　我们已经编写好了一个装饰器，但是当将它用在一个函数上时，一些重要的元数据比如函数名、文档字符串、函数注解以及调用签名都丢失了。

9.2.2　解决方案

　　每当定义一个装饰器时，应该总是记得为底层的包装函数添加functools库中的@wraps装饰器。示例如下：

```
import time

from functools import

wraps
def

timethis(func):
    '''
```

```
    Decorator that reports the execution time.

    '''

    @wraps(func)
    def

wrapper(*args, **kwargs):
    start = time.time()
    result = func(*args, **kwargs)
    end = time.time()
    print

(func.__name__, end-start)
    return

result
return

wrapper
```

下面是使用这个装饰器的示例，并且展示了如何检视结果函数的元数据：

```
>>> @timethis
... def

countdown(n:int):
```

```
...
    '''
...
    Counts down
...
    '''
...
    while
n > 0:
...
            n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>
```

9.2.3 讨论

编写装饰器的一个重要部分就是拷贝装饰器的元数据。如果忘记使用@wraps，就会发现被包装的函数丢失了所有有用的信息。例如，如果忽略@wraps，上面这个例子中的元数据看起来就是这样的：

```
>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>
```

@wraps装饰器的一个重要特性就是它可以通过__wrapped__属性来访问被包装的那个函数。例如，如果希望直接访问被包装的函数，则可以这样做：

```
>>> countdown.__wrapped__(100000)
>>>
```

__wrapped__属性的存在同样使得装饰器函数

可以合适地将底层被包装函数的签名暴露出来。例如：

```
>>> from inspect import


signature
>>> print


(signature(countdown))
(n:int)
>>>
```

常会提到的一个问题是如何让装饰器直接拷贝被包装的原始函数的调用签名（即，不使用*args和**kwargs）。一般来说，如果不采用涉及生成代码字符串和exec()的技巧，那么这很难实现。坦白地说，通常我们最好还是使用@wraps。这样可以依赖于一个事实，即，底层的函数签名可以通过访问__wrapped__属性来传递。有关函数签名的更多信息可以参阅9.16节。

9.3 对装饰器进行解包装

9.3.1 问题

我们已经把装饰器添加到一个函数上了，但是想“撤销”它，访问未经过包装的那个原始函数。

9.3.2 解决方案

假设装饰器的实现中已经使用了@wraps（参见9.2节），一般来说我们可以通过访问__wrapped__属性来获取对原始函数的访问。示例如下：

```
>>> @somedecorator
>>> def


add(x, y):
...


     return


x + y
...



>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
```

```
7
>>>
```

9.3.3　讨论

直接访问装饰器背后的那个未包装过的函数对于调试、反射（introspection，也有译为“自省”）以及其他一些涉及函数的操作是很有帮助的。但是，本节讨论的技术只有在实现装饰器时利用functools模块中的@wraps对元数据进行了适当的拷贝，或者直接设定了__wrapped__属性时才有用。

如果有多个装饰器已经作用于某个函数上了，那么访问__wrapped__属性的行为目前是未定义的，应该避免这种情况。在Python 3.3中，这么做会绕过所有的包装层。例如，假设有如下的代码：

```
from functools import


wraps

def


decorator1(func):
    @wraps(func)
    def


wrapper(*args, **kwargs):
```

```
        print


('Decorator 1')
        return


func(*args, **kwargs)
    return


wrapper

def


decorator2(func):
    @wraps(func)
    def


wrapper(*args, **kwargs):
        print


('Decorator 2')
        return


func(*args, **kwargs)
    return


wrapper

@decorator1
@decorator2
def


add(x, y):
    return


x + y
```

当调用装饰过的函数以及通过__wrapped__属性调用原始函数时就会出现这样的情况：

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>
```

然而，这种行为已经被报告为一个bug了（参见http://bugs.python.org/issue17482 ），可能会在今后释出的版本中修改为暴露出合适的装饰器链（decorator chain）。

最后但同样重要的是，请注意并不是所有的装饰器都使用了@wraps，因此有些装饰器的行为可能与我们期望的有所区别。特别是，由内建的装饰器@staticmethod和@classmethod创建的描述符（descriptor）对象并不遵循这个约定（相反，它们会把原始函数保存在__func__属性中）。所以，具体问题需要具体分析，每个人遇到的情况可能不同。

9.4 定义一个可接受参数的装饰器

9.4.1 问题

我们想编写一个可接受参数的装饰器函数。

9.4.2 解决方案

让我们用一个例子来说明接受参数的过程。假设我们想编写一个为函数添加日志功能的装饰器，但是又允许用户指定日志的等级以及一些其他的细节作为参数。下面是定义这个装饰器的可能做法：

```
from functools import


wraps
import logging




def


logged(level, name=None, message=None):
    '''



    Add logging to a function. level is the logging
```

```
    level, name is the logger name, and message is the

    log message. If name and message aren't specified,

    they default to the function's module and name.

    '''

    def

decorate(func):
        logname = name if

name else

func.__module__
        log = logging.getLogger(logname)
        logmsg = message if

message else

func.__name__

        @wraps(func)
        def

wrapper(*args, **kwargs):
            log.log(level, logmsg)
```

```
            return

func(*args, **kwargs)
        return

wrapper
    return

decorate

# Example use


@logged(logging.DEBUG)
def

add(x, y):
    return

x + y

@logged(logging.CRITICAL, 'example')
def

spam():
    print

('Spam!')
```

初看上去这个实现显得很有技巧性，但其中的思想相对来说是很简单的。最外层的logged()函数接受所需的参数，并让它们对装饰器的内层函数可

见。内层的decorate()函数接受一个函数并给它加上一个包装层。关键部分在于这个包装层可以使用传递给logged()的参数。

9.4.3 讨论

编写一个可接受参数的装饰器是需要一些技巧的，因为这会涉及底层的调用顺序。具体来说，如果有这样的代码：

```
@decorator(x, y, z)
def


func(a, b):
    pass
```

装饰的过程会按照下列方式来进行：

```
def


func(a, b):
    pass



func = decorator(x, y, z)(func)
```

请仔细观察，decorator(x, y, z)的结果必须是一个可调用对象，这个对象反过来接受一个函数作为输入，并对其进行包装。请参见9.7节中另一个关于让装饰器接受参数的例子。

9.5　定义一个属性可由用户修改的装饰器

9.5.1　问题

我们想编写一个装饰器来包装函数，但是可以让用户调整装饰器的属性，这样在运行时能够控制装饰器的行为。

9.5.2　解决方案

下面给出的解决方案对上一节的示例进行了扩展，引入了访问器函数（accessor function），通过使用nonlocal关键字声明变量来修改装饰器内部的属性。之后把访问器函数作为函数属性附加到包装函数上。

```
from functools import


wraps, partial
import logging



# Utility decorator to attach a function as an attribute of obj
```

```
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    '''
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
```

```
    '''

    def

decorate(func):
        logname = name if

name else

func.__module__
        log = logging.getLogger(logname)
        logmsg = message if

message else

func.__name__

        @wraps(func)
        def

wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return

func(*args, **kwargs)

        # Attach setter functions



        @attach_wrapper(wrapper)
        def
```

```
set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def


set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return


wrapper
    return


decorate

# Example use



@logged(logging.DEBUG)
def


add(x, y):
    return


x + y

@logged(logging.CRITICAL, 'example')
def


spam():
    print


('Spam!')
```

下面的交互式会话展示了在完成上面的定义之后对各项属性的修改：

```
>>> import logging



>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
5

>>> # Change the log message



>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5

>>> # Change the log level



>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>
```

9.5.3 讨论

本节示例的关键就在访问器函数（即，

set_message()和set_level()），它们以属性的形式附加到了包装函数上。每个访问器函数允许对nonlocal变量赋值来调整内部参数。

这个示例中有一个令人惊叹的特性，那就是访问器函数可以跨越多个装饰器层进行传播（如果所有的装饰器都使用了@functools.wraps 的话）。例如，假设引入了一个额外的装饰器，比如9.2节中的@timethis，然后编写了如下的代码：

```
@timethis
@logged(logging.DEBUG)
def


countdown(n):
    while


n > 0:
        n -= 1
```

就会发现访问器函数依然可以正常工作：

```
>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>
```

如果把装饰器的顺序像下面这样颠倒一下，就会发现访问器函数还是能够以相同的方式工作。

```
@logged(logging.DEBUG)
@timethis
def


countdown(n):
    while


n > 0:
    n -= 1
```

尽管这里没有给出，我们也可以通过添加如下额外的代码来实现用访问器函数返回内部的状态值：

```
...
@attach_wrapper(wrapper)
def


get_level():
    return


level

# Alternative
```

```
wrapper.get_level = lambda


: level
...
```

本节中一个极为微妙的地方在于为什么要在一开始使用访问器函数。比方说，我们可能会考虑其他的方案，完全基于对函数属性的直接访问，示例如下：

```
...
@wraps(func)
def


wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return


func(*args, **kwargs)

# Attach adjustable attributes



wrapper.level = level
wrapper.logmsg = logmsg
wrapper.log = log
...
```

这种方法只能用在最顶层的装饰器上。如果在

当前顶层的装饰器上又添加了一个装饰器（比如示例中的@timethis），这样就会隐藏下层的属性使得它们无法被修改。而使用访问器函数可以绕过这个限制。

最后但同样重要的是，本节展示的解决方案可以作为类装饰器的一种替代方式，我们在9.9节中会继续谈到相关的主题。

9.6 定义一个能接收可选参数的装饰器

9.6.1 问题

我们想编写一个单独的装饰器，使其既可以像@decorator这样不带参数使用，也可以像@decorator(x, y, z)这样接收可选参数。但是，由于简单的装饰器和可接收参数的装饰器之间存在不同的调用约定（calling convention），这样看来似乎并没有直接的方法来处理。

9.6.2 解决方案

我们对9.5节中记录日志的代码做了修改，定义了一个可接受可选参数的装饰器：

```
from functools import

wraps, partial
import logging

def
```

```
logged(func=None, *, level=logging.DEBUG, name=None, message=None
    if

func is

None:
        return

partial(logged, level=level, name=name, message=message)

    logname = name if

name else

func.__module__
    log = logging.getLogger(logname)
    logmsg = message if

message else

func.__name__
    @wraps(func)
    def

wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return

func(*args, **kwargs)
    return

wrapper

# Example use
```

```
@logged
def

add(x, y):
    return

x + y

@logged(level=logging.CRITICAL, name='example')
def

spam():
    print

('Spam!')
```

从示例中可以看到，现在这个装饰器既能够以简单的形式（即@logged）使用，也可以提供可选的参数给它（即，@logged(level=logging.CRITICAL, name='example')）。

9.6.3 讨论

本节提到的实际上是一种编程一致性（programming consistency）的问题。当使用装饰器时，大部分程序员习惯于完全不使用任何参数，

或者就像示例中那样使用参数。从技术上来说，如果装饰器的所有参数都是可选的，那么可以像这样来使用：

```
@logged()
def


add(x, y):
    return


 x+y
```

但是这和我们常见的形式不太一样，如果程序员忘记加上那个额外的圆括号就可能会导致常见的使用错误。本节提到的技术可以让装饰器以一致的方式使用，既可以带括号也可以不带括号。

要理解代码是如何工作的，就需要对装饰器是如何施加到函数上，以及对它们的调用约定有着透彻的理解才行。以一个简单的装饰器为例：

```
# Example use



@logged
def


add(x, y):
```

```
    return

x + y
```

调用顺序是这样的：

```
def

add(x, y):
    return

x + y
add = logged(add)
```

在这种情况下，被包装的函数只是作为第一个参数简单地传递给logged。因而，在解决方案中，logged()的第一个参数就是要被包装的那个函数。其他所有的参数都必须有一个默认值。

对于一个可接受参数的装饰器，例如：

```
@logged(level=logging.CRITICAL, name='example')
def

spam():
    print
```

```
('Spam!')
```

其调用顺序是这样的：

```
def

spam():
    print

('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

在初次调用logged()时，被包装的函数并没有传递给logged。因此在装饰器中，被包装的函数必须作为可选参数。这样一来，反过来迫使其他的参数都要通过关键字来指定。此外，当传递了参数后装饰器应该返回一个新函数，要包装的函数就作为参数传递给这个新函数（见9.5节）。要做到这一点，我们在解决方案中利用functools.partial来实现这个聪明的技巧。具体来说，它只是返回了一个部分完成的版本，除了要被包装的函数之外，其他所有的参数都已经确定好了。关于partial()的使用，可参阅7.8节以获取更多的细节。

9.7　利用装饰器对函数参数强制执行类型检查

9.7.1　问题

我们想为函数参数添加强制性的类型检查功能，将其作为一种断言或者与调用者之间的契约。

9.7.2　解决方案

在给出解决方案代码之前，本节的目标是提供一种手段对函数的输入参数类型做强制性的类型检查。下面这个简短的示例说明了这种思想：

```
>>> @typeassert(int, int)
... def


add(x, y):
...


     return


x + y
...



>>>
```

```
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

现在，让我们看看装饰器@typeassert的实现：

```
from inspect import


signature
from functools import


wraps

def


typeassert(*ty_args, **ty_kwargs):
    def


decorate(func):
        # If in optimized mode, disable type checking



        if not


__debug__:
            return


func
```

```
        # Map function argument names to supplied types



        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arg

        @wraps(func)
        def


wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments



            for


name, value in


bound_values.arguments.items():
                 if


name in


bound_types:
                     if not


isinstance(value, bound_types[name]):
                       raise TypeError


(
                         'Argument {} must be {}'.format(name, bo
                         )
            return
```

```
func(*args, **kwargs)
        return


wrapper
    return


decorate
```

我们会发现这个装饰器相当灵活，既允许指定函数参数的所有类型，也可以只指定一部分子集。此外，类型既可以通过位置参数来指定，也可以通过关键字参数来指定。示例如下：

```
>>> @typeassert(int, z=int)
... def


spam(x, y, z=42):
...


     print


(x, y, z)
...



>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>
```

9.7.3　讨论

本节展示了一个高级的装饰器例子，引入了一些重要且有用的概念。

首先，装饰器的一个特性就是它们只会在函数定义的时候应用一次。在某些情况下，我们可能想禁止由装饰器添加的功能。为了做到这点，只要让装饰器函数返回那个未经过包装的函数即可。在解决方案中，如果全局变量__debug__被设为False，下列代码就会返回未修改过的函数（当Python解释器以-O或-OO的优化模式执行的话，则属于这种情况）。

```
...
def

decorate(func):
    # If in optimized mode, disable type checking

    if not

__debug__:
        return
```

```
func
    ...
```

接下来，编写这个装饰器比较棘手的地方在于要涉及对被包装函数的参数签名做检查。在这里，我们可选择的工具应该是inspect.signature()函数。简单来说，这个函数允许我们从一个可调用对象中提取出参数签名信息。示例如下：

```
>>> from inspect import


signature
>>> def


spam(x, y, z=42):
...



     pass




...




>>> sig = signature(spam)
>>> print


(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
```

```
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x100
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

在装饰器实现的第一部分中，我们使用签名的bind_partial()方法来对提供的类型到参数名做部分绑定。下面的示例说明了其中发生了些什么：

```
>>> bound_types = sig.bind_partial(int,z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

在这个部分绑定中，我们会注意到缺失的参数被简单地忽略掉了（即，这里没有对参数 y 做绑定）。但是，绑定过程中最重要的部分就是创建了有序字典 bound_types. arguments。这个字典将参数名以函数签名中相同的顺序映射到所提供的值上。在我们的装饰器中，这个映射包含了我们打算强制施行的类型断言。

在由装饰器构建的包装函数中用到了sig.bind()

方法。bind()就如同bind_partial()一样，只是它不允许出现缺失的参数。因此，下面的示例中必须给出所有的参数：

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

利用这个映射，要强制施行断言相对来说就很简单了：

```
>>> for

name, value in

bound_values.arguments.items():
...

     if

name in

bound_types.arguments:
...

         if not

isinstance(value, bound_types.arguments[name]):
...
```

```
            raise TypeError

()
...


>>>
```

解决方案中一个多少有些微妙的地方是，对于具有默认值的参数，如果未提供参数，则断言机制不会作用在其默认值上。例如，下面的代码可以工作，即使items的默认值是“错误”的类型：

```
>>> @typeassert(int, list)
... def

bar(x, items=None):
...

    if

items is

None:
...

        items = []
...
```

```
    items.append(x)
...


    return


items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

最后一点关于设计上的讨论应该就是装饰器参数与函数注解（function annotation）的对比了。例如，为什么不把装饰器实现为检查函数注解呢？

```
@typeassert
def


spam(x:int, y, z:int = 42):
    print


(x,y,z)
```

不使用函数注解的一个可能原因在于函数的每

个参数只能赋予一个单独的注解。因此，如果把注解用于类型断言，则它们就不能用在别处了。此外，装饰器@typeassert不能用于使用了注解的函数还有另一个原因。如同解决方案中展示的那样，通过使用装饰器参数，这个装饰器变得更加通用了，可以用于任何函数——即使是使用了注解的函数也是如此。

关于函数签名对象的更多信息可以在PEP 362（http://www.python.org/dev/peps/ pep-0362 ）以及inspect模块的文档（http://docs.python.org/3/library/inspect.html ）中找到。9.16节中也有一个额外的示例可供参考。

9.8　在类中定义装饰器

9.8.1　问题

我们想在类中定义一个装饰器，并将其作用于其他的函数或者方法上。

9.8.2　解决方案

在类中定义一个装饰器是很直接的，但是首先我们需要理清装饰器将以什么方式来应用。具体来说就是以实例方法还是以类方法的形式应用。下面的示例说明了这些区别：

```
from functools import

wraps
class A

:
    # Decorator as an instance method

    def

decorator1(self, func):
```

```
        @wraps(func)
        def


wrapper(*args, **kwargs):
            print


('Decorator 1')
            return


func(*args, **kwargs)
        return


wrapper

    # Decorator as a class method



    @classmethod
    def


decorator2(cls, func):
        @wraps(func)
        def


wrapper(*args, **kwargs):
             print


('Decorator 2')
             return


func(*args, **kwargs)
    return


wrapper
```

下面的示例展示了这两种装饰器会如何应用：

```
# As an instance method

a = A()

@a.decorator1
def

spam():
    pass

# As a class method

@A.decorator2
def

grok():
    pass
```

如果观察得够仔细，就会发现其中一个装饰器来自于实例a，而另一个装饰器来自于类A。

9.8.3 讨论

在类中定义装饰器乍看起来可能有些古怪，但是在标准库中也可以找到这样的例子。尤其是，内建的装饰器@property实际上是一个拥有getter()、setter()和deleter()方法的类，每个方法都可作为一个装饰器。示例如下：

```
class Person

:
    # Create a property instance


    first_name = property()

    # Apply decorator methods


    @first_name.getter
    def


first_name(self):
        return


self._first_name

    @first_name.setter
    def


first_name(self, value):
        if not
```

```
isinstance(value, str):
            raise TypeError

('Expected a string')
        self._first_name = value
```

至于为什么要定义成这种形式，关键原因在于这里的多个装饰器方法都在操纵property实例的状态。因此，如果需要装饰器在背后记录或合并信息，这就是个很明智的方法。

当编写类中的装饰器时，一个常见的困惑就是如何在装饰器代码中恰当地使用 self 或cls参数。尽管最外层的装饰器函数比如decorator1()或decorator2()需要提供一个self或cls参数（因为它们是类的一部分），但内层定义的包装函数一般不需要包含额外的参数。这就是为什么示例中两个装饰器创建的wrapper()函数并没有包含self参数的原因。唯一一种可能会用到这个参数的场景就是需要在包装函数中访问实例的某个部分。否则，就不必为此操心。

关于把装饰器定义在类的内部，还有最后一个微妙的方面需要考虑。那就是它们在继承中的潜在用途。例如，假设想把定义在类A中的装饰器施加

于定义在子类B中的方法上。要做到这点，需要像这样编写代码：

```
class B


(A):
    @A.decorator2
    def


bar(self):
        pass
```

特别是，这里的装饰器必须定义为类方法，而且使用时必须显式地给出父类A的名称。不能使用像@B.decoator2这样的名称，因为在定义该方法的时候类B根本就没有创建出来。

9.9　把装饰器定义成类

9.9.1　问题

　　我们想用装饰器来包装函数，但是希望得到的结果是一个可调用的实例。我们需要装饰器既能在类中工作，也可以在类外部使用。

9.9.2　解决方案

　　要把装饰器定义成类实例，需要确保在类中实现__call__()和__get__()方法。例如，下面的代码定义了一个类，可以在另一个函数上添加一个简单的性能分析层：

```
import types

from functools import

wraps
class Profiled

:
    def
```

```
__init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def


__call__(self, *args, **kwargs):
        self.ncalls += 1
        return


self.__wrapped__(*args, **kwargs)

    def


__get__(self, instance, cls):
        if


instance is


None:
            return


self
        else


:
            return


types.MethodType(self, instance)
```

要使用这个类，可以像一个普通的装饰器一样，要么在类中要么在类外部使用：

```
@Profiled
def


add(x, y):
    return


x + y

class Spam


:
    @Profiled
    def


bar(self, x):
        print


(self, x)
```

下面的交互式会话展示了这些函数是如何工作的：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
```

```
>>> Spam.bar.ncalls
3
```

9.9.3 讨论

把装饰器定义成类通常是简单明了的。但是，这里有一些相当微妙的细节值得做进一步的解释，尤其是计划将装饰器应用在实例的方法上时。

首先，这里对functools.wraps()函数的使用和在普通装饰器中的目的一样——意在从被包装的函数中拷贝重要的元数据到可调用实例中。

其次，解决方案中所展示的__get__()方法常常容易被忽视。如果省略掉__get__()并保留其他所有的代码，会发现当尝试调用被装饰的实例方法时会出现怪异的行为。例如：

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...



TypeError: spam() missing 1 required positional argument: 'x'
```

出错的原因在于每当函数实现的方法需要在类

中进行查询时，作为描述符协议（descriptor protocol）的一部分，它们的__get__()方法都会被调用，这部分内容在8.9节中已描述过。在这种情况下，__get__()的目的是用来创建一个绑定方法对象（最终会给方法提供self参数）。下面的例子说明了其中的机理：

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

在本节中，__get__()方法在这里确保了绑定方法对象会恰当地创建出来。type. MethodType()手动创建了一个绑定方法在这里使用。绑定方法只会在使用到实例的时候才创建。如果在类中访问该方法，__get__()的instance参数就设为None，直接返回Profiled实例本身。这样就使得获取实例的ncalls

属性成为可能。

如果想在某些方面避免这种混乱，可以考虑装饰器的替代方案，即9.5节中描述过的利用闭包和nonlocal变量。示例如下：

```
import types


from functools import


wraps

def


profiled(func):
    ncalls = 0
    @wraps(func)
    def


wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return


func(*args, **kwargs)
    wrapper.ncalls = lambda


: ncalls
    return


wrapper
```

```
# Example



@profiled
def


add(x, y):
    return


x + y
```

这个例子使用起来和之前的方案几乎一致，除了现在访问ncalls时是以函数属性的形式来进行。示例如下：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

9.10 把装饰器作用到类和静态方法上

9.10.1 问题

我们想在类或者静态方法上应用装饰器。

9.10.2 解决方案

将装饰器作用到类和静态方法上是简单而直接的，但是要保证装饰器在应用的时候需要放在@classmethod和@staticmethod之前。示例如下：

```
import time

from functools import

wraps
# A simple decorator

def

timethis(func):
```

```
    @wraps(func)
    def


wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print


(end-start)
        return


r
    return


wrapper

# Class illustrating application of the decorator to different ki



class Spam


:
    @timethis
    def


instance_method(self, n):
        print


(self, n)
        while


n > 0:
            n -= 1

    @classmethod
```

```
    @timethis
    def


class_method(cls, n):
        print


(cls, n)
        while


n > 0:
            n -= 1

    @staticmethod
    @timethis
    def


static_method(n):
        print


(n)
        while


n > 0:
            n -= 1
```

上面代码中的类和静态方法应该能够正常工作，此外还为它们添加了额外的计时功能：

```
>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
```

```
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>
```

9.10.3 讨论

如果装饰器的顺序搞错了，那么将得到错误提示。例如，如果像下面这样使用装饰器：

```
class Spam


:
    ...
    @timethis
    @staticmethod
    def


static_method(n):
        print


(n)
        while


n > 0:
            n -= 1
```

这样的话，调用static_method将崩溃：

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "timethis.py", line 6, in wrapper
    start = time.time()
TypeError: 'staticmethod' object is not callable
>>>
```

这里的问题在于@classmethod和@staticmethod并不会实际创建可直接调用的对象。相反，它们创建的是特殊的描述符对象（参见8.9节中对描述符的讲解）。因此，如果尝试在另一个装饰器中像函数那样使用它们，装饰器就会崩溃。确保这些装饰器出现在@classmethod和@staticmethod之前就能解决这个问题。

本节提到的技术有一个至关重要的应用场景，那就是在抽象基类中定义类方法和静态方法，这也是在8.12节中谈到的主题。例如，如果想定义一个抽象类方法，可以使用下面的代码来完成：

```
from abc import


ABCMeta, abstractmethod

class A


(metaclass=ABCMeta):
    @classmethod
```

```
    @abstractmethod
    def


method(cls):
    pass
```

在上述代码中，@classmethod和@abstractmethod出现的顺序是很重要的。如果将这两个装饰器调换一下位置，那么就会产生崩溃。

9.11 编写装饰器为被包装的函数添加参数

9.11.1 问题

我们想编写一个装饰器为被包装的函数添加额外的参数。但是，添加的参数不能影响到该函数已有的调用约定。

9.11.2 解决方案

可以使用keyword-only参数将额外的参数注入到函数的调用签名中。考虑如下的装饰器：

```
from functools import

wraps

def

optional_debug(func):
    @wraps(func)
    def

wrapper(*args, debug=False, **kwargs):
        if
```

```
debug:
            print

('Calling', func.__name__)
        return

func(*args, **kwargs)
    return

wrapper
```

下面的示例展示了装饰器是如何工作的:

```
>>> @optional_debug
... def

spam(a,b,c):
...

     print

(a,b,c)
...


>>> spam(1,2,3)
1 2 3
>>> spam(1,2,3, debug=True)
Calling spam
1 2 3
>>>
```

9.11.3 讨论

为被包装的函数添加额外的参数并不是装饰器最常见的用法。但是，对于避免某些特定的代码重复模式来说是一项有用的技术。例如，如果有一些这样的代码：

```
def a(x, debug=False):
    if debug:
        print('Calling a')
    ...

def b(x, y, z, debug=False):
    if debug:
        print('Calling b')
    ...

def
```

```
c(x, y, debug=False):
    if

debug:
        print

('Calling c')
    ...
```

可以将这些代码重构为如下形式：

```
@optional_debug
def

a(x):
    ...

@optional_debug
def

b(x, y, z):
    ...

@optional_debug
def

c(x, y):
    ...
```

本节给出的实现依赖于这样一个事实，即

keyword-only 参数可以很容易地添加到那些以*args和**kwargs作为形参的函数上。keyword-only参数会作为特殊情况从随后的调用中挑选出来，调用函数时只会使用剩下的位置参数和关键字参数。

这里有个棘手的问题需要考虑。在添加的参数和被包装函数的参数之间可能会出现潜在的名称冲突问题。例如，如果把@optional_debug装饰器作用到一个已经把debug作为参数的函数上，此时就会出错。要解决这个问题，需要添加额外的检查：

```
from functools import

wraps
import inspect

def

optional_debug(func):
    if

'debug' in

inspect.getargspec(func).args:
        raise TypeError

('debug argument already defined')
```

```
    @wraps(func)
    def


wrapper(*args, debug=False, **kwargs):
        if


debug:
            print


('Calling', func.__name__)
        return


func(*args, **kwargs)
    return


wrapper
```

本节中最后一个需要考虑修改的地方在于如何恰当地管理函数签名。精明的程序员会意识到被包装函数的签名是错误的。例如：

```
>>> @optional_debug
... def


add(x,y):
...


     return


x+y
...
```

```
>>> import inspect

>>> print

(inspect.signature(add))
(x, y)
>>>
```

这可以通过如下的修改来解决：

```
from functools import

wraps
import inspect

def

optional_debug(func):
    if

'debug' in

inspect.getargspec(func).args:
        raise TypeError

('debug argument already defined')
```

```
    @wraps(func)
    def


wrapper(*args, debug=False, **kwargs):
        if


debug:
            print


('Calling', func.__name__)
        return


func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                                    inspect.Parameter.KEYWORD_ONl
                                    default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return


wrapper
```

修改之后，现在包装函数的签名就能正确反映出debug参数了。示例如下：

```
>>> @optional_debug
... def


add(x,y):
...
```

```
    return

x+y
...

>>> print

(inspect.signature(add))
(x, y, *, debug=False)
>>> add(2,3)
5
>>>
```

要获得更多有关函数签名方面的信息，可参阅9.16节。

9.12 利用装饰器给类定义打补丁

9.12.1 问题

我们想检查或改写一部分类的定义，以此来修改类的行为，但是不想通过继承或者元类的方式来做。

9.12.2 解决方案

对于类装饰器来说这是绝佳的应用场景。比方说，下面有一个类装饰器重写了__getattribute__特殊方法，为其加上了日志记录功能。

```
def

log_getattribute(cls):
    # Get the original implementation

    orig_getattribute = cls.__getattribute__

    # Make a new definition

    def
```

```
new_getattribute(self, name):
        print

('getting:', name)
        return

orig_getattribute(self, name)

    # Attach to the class and return

    cls.__getattribute__ = new_getattribute
    return

cls

# Example use

@log_getattribute
class A

:
    def

__init__(self,x):
        self.x = x
    def

spam(self):
        pass
```

如果试着使用解决方案中给出的类，就会得到以下结果：

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

9.12.3 讨论

类装饰器常常可以直接作为涉及混合类（mixin）或者元类等高级技术的替代方案。例如，对于解决方案中的例子，另一种可选的实现方法是使用继承：

```
class LoggedGetattribute


:
    def


__getattribute__(self, name):
        print


('getting:', name)
        return
```

```
super().__getattribute__(name)

# Example:



class A


(LoggedGetattribute):
    def


__init__(self,x):
        self.x = x
    def


spam(self):
        pass
```

这么做可行，但是要想理解其中的原理，则必须对方法解析顺序（MRO）、super()以及其他有关继承方面的知识有所了解（详见8.7节）。从某种意义上说，类装饰器这种解决方案要更加直接，而且不会在继承体系中引入新的依赖关系。事实证明，由于不依赖对super()函数的使用，运行速度也会稍快一些。

如果要将多个类装饰器作用于某个类之上，那么可能需要考虑添加的顺序问题。例如，如果某个

装饰器是用全新的实现来替换一个类方法，而另一个装饰器只是对已有的方法做包装，添加一些额外的逻辑处理，那么很可能需要先将第一个装饰器作用于类上。

请参考8.13节中另一个关于类装饰器的示例。

9.13　利用元类来控制实例的创建

9.13.1　问题

我们想改变实例创建的方式，以此来实现单例模式、缓存或者其他类似的特性。

9.13.2　解决方案

作为Python程序员，大家都应该知道如果定义了一个类，那么创建实例时就好像在调用一个函数一样。示例如下：

```
class Spam



:
    def



__init__(self, name):
        self.name = name

a = Spam('Guido')
b = Spam('Diana')
```

如果想定制化这个步骤，则可以通过定义一个

元类并以某种方式重新实现它的__call__()方法。为了说明这个过程，假设我们不想让任何人创建出实例：

```
class NoInstances


(type):
    def


__call__(self, *args, **kwargs):
        raise TypeError


("Can't instantiate directly")

# Example



class Spam


(metaclass=NoInstances):
    @staticmethod
    def


grok(x):
        print


('Spam.grok')
```

在这种情况下，用户可以调用定义的静态方法，但是没法以普通的方式创建出实例。示例如

下：

```
>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example1.py", line 7, in __call__
    raise TypeError


("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>
```

现在，假设我们想实现单例模式（即，这个类只能创建唯一的一个实例）。相对来说这就很直接了，示例如下：

```

class Singleton


(type):
    def


__init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def


__call__(self, *args, **kwargs):
        if
```

```
self.__instance is


None:
            self.__instance = super().__call__(*args, **kwargs)
            return


self.__instance
        else


:
            return


self.__instance
# Example



class Spam


(metaclass=Singleton):
    def


__init__(self):
        print


('Creating Spam')
```

在这种情况下，这个类只能创建出唯一的实例。示例如下：

```
>>> a = Spam()
Creating Spam
>>> b = Spam()
```

```
>>> a is


b
True
>>> c = Spam()
>>> a is


c
True
>>>
```

最后，假设我们想创建缓存实例（cached instance，8.25节有介绍）。我们用一个元类来实现：

```
import weakref




class Cached


(type):
    def


__init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def


__call__(self, *args):
        if
```

```
args in

self.__cache:
            return

self.__cache[args]
        else

:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return

obj

# Example

class Spam

(metaclass=Cached):
    def

__init__(self, name):
        print

('Creating Spam({!r})'.format(name))
        self.name = name
```

下面的交互式会话展示了这个类的行为：

```
>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido')         # Cached



>>> a is


b
False
>>> a is


c                     # Cached value returned



True
>>>
```

9.13.3 讨论

通过元类来实现各种创建实例的模式常常比那些不涉及元类的解决方案要优雅。如果不用元类，那就得将类隐藏在某种额外的工厂函数之后。例如，要实现单例模式，可能会用到下面这种技巧：

```
class _Spam


:
    def
```

```
__init__(self):
        print


('Creating Spam')

_spam_instance = None
def


Spam():
    global


_spam_instance
    if


_spam_instance is not


None:
        return


_spam_instance
    else


:
        _spam_instance = _Spam()
        return


_spam_instance
```

尽管使用元类的解决方案涉及许多更加高级的概念，但最终的代码看起来会更加清晰，也没有那

么多所谓的技巧。

参见8.25节以得到更多关于创建缓存实例、弱引用（weak reference）以及其他细节方面的信息。

9.14 获取类属性的定义顺序

9.14.1 问题

我们想自动记录下属性和方法在类中定义的顺序，这样就能利用这个顺序来完成各种操作（例如序列化处理、将属性映射到数据库中等）。

9.14.2 解决方案

要获取类定义体中的有关信息，可以通过元类来轻松实现。在下面的示例中，元类使用OrderedDict（有序字典）来获取描述符的定义顺序：

```
from collections import

OrderedDict
# A set of descriptors for various types

class Typed

:
    _expected_type = type(None)
    def
```

```
__init__(self, name=None):
        self._name = name

    def


__set__(self, instance, value):
        if not


isinstance(value, self._expected_type):
            raise TypeError


('Expected ' + str(self._expected_type))
        instance.__dict__[self._name] = value

class Integer


(Typed):
    _expected_type = int

class Float


(Typed):
    _expected_type = float

class String


(Typed):
    _expected_type = str

# Metaclass that uses an OrderedDict for class body



    class OrderedMeta


(type):
```

```
        def


__new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for


name, value in


clsdict.items():
            if


isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return


type.__new__(cls, clsname, bases, d)
    @classmethod
    def


__prepare__(cls, clsname, bases):
        return


OrderedDict()
```

在这个元类中，描述符的定义顺序是通过使用OrderedDict在执行类的定义体时获取到的。得到的结果会从字典中提取出来然后保存到类的属性_order中。这之后，类方法能够以各种方式使用属

性_order。例如，下面这个简单的类利用这个顺序实现了一个方法，用来将实例数据序列化为一行CSV数据：

```
class Structure


(metaclass=OrderedMeta):
    def


as_csv(self):
        return


','.join(str(getattr(self,name)) for


name in


self._order)
# Example use



class Stock


(Structure):
    name = String()
    shares = Integer()
    price = Float()
    def


__init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

下面的交互式会话展示了如何使用例子中的Stock类：

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.as_csv()
'GOOG,100,490.1'
>>> t = Stock('AAPL','a lot', 610.23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>
```

9.14.3 讨论

本节的全部核心就在__prepare__()方法上，该特殊方法定义在元类OrderedMeta中。该方法会在类定义一开始的时候立刻得到调用，调用时以类名和基类名称作为参数。它必须返回一个映射型对象（mapping object）供处理类定义体时使用。由于返回的是OrderedDict实例而不是普通的字典，因此类中各个属性间的顺序就可以方便地得到维护。

如果想使用自定义的字典型对象，那么对上述功能进行扩展也是有可能的。例如，考虑下面这个

解决方案，它可以拒绝类中出现重复的定义：

```
from collections import


OrderedDict

class NoDupOrderedDict


(OrderedDict):
    def


__init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def


__setitem__(self, name, value):
        if


name in


self:
            raise TypeError


('{} already defined in {}'.format(name, self.clsname))
        super().__setitem__(name, value)

class OrderedMeta


(type):
    def


__new__(cls, clsname, bases, clsdict):
```

```
        d = dict(clsdict)
        d['_order'] = [name for


name in


clsdict if


name[0] != '_']
        return


type.__new__(cls, clsname, bases, d)

    @classmethod
    def


__prepare__(cls, clsname, bases):
        return


NoDupOrderedDict(clsname)
```

如果使用这个元类，并且创建一个类让它拥有重复的属性，看看会发生什么吧：

```
>>> class A


(metaclass=OrderedMeta):
...


     def
```

```
spam(self):
...
            pass
...
    def
spam(self):
...
            pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in A
  File "dupmethod2.py", line 25, in __setitem__
    (name, self.clsname))
TypeError: spam already defined in A
>>>
```

本节中最后一个需要考虑的重要部分是在元类的__new__()方法中对自定义的字典应该如何处理。尽管我们在类定义中使用的是其他形式的字典，当创建最终的类对象时，还是需要将这个字典

转换为一个合适的dict实例才行。这正是d = dict(clsdict)这行代码的目的所在。

能够获取到类属性的定义顺序看起来似乎微不足道，但对于某些特定类型的应用来说却是非常重要的功能。例如，在一个对象关系映射器（ORM）中，类的编写方式可能同示例中展示的那样很相似：

```
class Stock


(Model):
    name = String()
    shares = Integer()
    price = Float()
```

而在底层，可能想获取到属性定义的顺序，以此将对象映射到数据库表项中的元组或者行上（即，和示例中as_csv()方法实现的功能类似）。本节给出的解决方案是非常直截了当的，而且通常情况下比其他可选的方法要更简单（一般会通过在描述符类中维护一个隐藏的计数器来实现）。

9.15　定义一个能接受可选参数的元类

9.15.1　问题

我们想定义一个元类，使得在定义类的时候能够提供可选的参数。这样的话在创建类型的时候可以对处理过程进行控制或配置。

9.15.2　解决方案

在定义类的时候，Python允许我们在class语句中通过使用metaclass关键字参数来指定元类。例如，在抽象基类中我们可以这样指定元类：

```
from abc import


ABCMeta, abstractmethod

class IStream


(metaclass=ABCMeta):
    @abstractmethod
    def


read(self, maxsize=None):
```

```
        pass



    @abstractmethod
    def


write(self, data):
        pass
```

但是，在自定义的元类中我们还可以提供额外的关键字参数，就像这样：

```
class Spam


(metaclass=MyMeta, debug=True, synchronize=True):
    ...
```

要在元类中支持这样的关键字参数，需要保证在定义__prepare__()、__new__()以及__init__()方法时使用keyword-only参数来指定它们，就像下面这样：

```

```

```
class MyMeta

(type):
    # Optional



    @classmethod
    def

__prepare__(cls, name, bases, *, debug=False, synchronize=False)
        # Custom processing



        ...
        return

super().__prepare__(name, bases)

    # Required



    def

__new__(cls, name, bases, ns, *, debug=False, synchronize=False)
        # Custom processing



        ...
        return

super().__new__(cls, name, bases, ns)

    # Required
```

```
    def

__init__(self, name, bases, ns, *, debug=False, synchronize=False
        # Custom processing

        ...
        super().__init__(name, bases, ns)
```

9.15.3 讨论

要对元类添加可选的关键字参数，需要理解类创建过程中所涉及的所有步骤。这是因为额外的参数会传递给每一个与该过程相关的方法。__prepare__()方法是第一个被调用的，用来创建类的名称空间，这是在处理类的定义体之前需要完成的。一般来说，这个方法只是简单地返回一个字典或者其他的映射型对象。__new__()方法用来实例化最终得到的类型对象，它会在类的定义体被完全执行完毕后才调用。最后调用的是__init__()方法，用来执行任何其他额外的初始化步骤。

当编写元类时，比较常见的做法是只定义一个__new__()或者__init__()方法，而不会同时定义这两者。但是，如果打算接受额外的关键字参数，那么这两个方法都必须提供，并且要提供可兼容的函数签名。默认的__prepare__()方法可接受任意的关

键字参数，只是会忽略它们。唯一一种需要自行定义__prepare__()方法的情况就是当额外的参数多少会影响到名称空间的创建管理时。

本节中使用了keyword-only参数，这也反映出一个事实，即这样的参数在创建类的过程中只会以关键字形式提供。

用关键字参数来配置元类也可以看做是通过类变量来实现同一目标的另一种方式。例如我们也可以这样实现对类的配置：

```
class Spam

(metaclass=MyMeta):
    debug = True
    synchronize = True
    ...
```

通过提供额外参数的方式来实现，这么做的优点在于它们不会污染类的名称空间。因为这些参数只对于类的创建而言有意义，对于类中需要执行的语句来说是没有实际意义的。此外，它们对于__prepare__()方法来说是可见的，而该方法会在处理类定义体中任何语句之前先得到运行。而另一方面，类变量只能被元类的__new__()和__init__()方

法访问。

9.16　在*args和**kwargs上强制规定一种参数签名

9.16.1　问题

我们已经编写了一个使用*args和**kwargs作为参数的函数或者方法，这样使得函数成为通用型的（即，可接受任意数量和类型的参数）。但是我们也想对传入的参数做检查，看看它们是否匹配了某个特定的函数调用签名。

9.16.2　解决方案

任何关于操作函数调用签名的问题，都应该使用inspect模块中的相应功能。这里我们尤其感兴趣的是Signature和Parameter这两个类。下面用一个交互式的例子来说明如何创建一个函数签名：

```
>>> from inspect import


Signature, Parameter
>>> # Make a signature for a func(x, y=42, *, z=None)



>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),
...
```

```
            Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, defaul
...


            Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]
>>> sig = Signature(parms)
>>> print


(sig)
(x, y=42, *, z=None)
>>>
```

一旦有了签名对象，就可以通过对象的bind()方法轻松将其绑定到*args和**kwargs上。示例如下：

```
>>> def


func(*args, **kwargs):
...


     bound_values = sig.bind(*args, **kwargs)
...


    for


name, value in


bound_values.arguments.items():
...
```

```
        print

(name,value)
...

>>> # Try various examples

>>> func(1, 2, z=3)
x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...

  File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bin
    raise TypeError

('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...

  File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bin
```

```
    raise TypeError

(msg) from None

TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...

  File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
    '{arg!r}'.format(arg=param.name))
TypeError: multiple values for argument 'x'
>>>
```

可以看到，将签名对象绑定到传入的参数上会强制施行所有常见的函数调用规则，包括要求必传的参数（例子中为x）、默认值、重复的参数等。

关于强制施行函数签名，这里有一个更为具体的例子。在代码中，基类定义了一个极其通用的__init__()方法，但是子类只提供一种期望接受的签名形式。

```
from inspect import

Signature, Parameter

def
```

```
make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for

name in

names]
    return

Signature(parms)

class Structure

:
    __signature__ = make_sig()
    def

__init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for

name, value in

bound_values.arguments.items():
            setattr(self, name, value)

# Example use

class Stock

(Structure):
    __signature__ = make_sig('name', 'shares', 'price')
```

```
class Point


(Structure):
    __signature__ = make_sig('x', 'y')
```

下面的交互式会话说明了Stock类是如何工作的：

```
>>> import inspect



>>> print


(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...



TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...



TypeError: multiple values for argument 'shares'
>>>
```

9.16.3 讨论

当需要编写通用型的库、编写装饰器或者实现代理时，使用形参为*args和**kwargs的函数是非常常见的。但是，这种函数的一个缺点就是如果想实现自己的参数检查机制，代码很快就会变的笨拙而混乱。这方面的例子可参考8.11节。使用签名对象则能简化这个步骤。

在解决方案的最后一个例子中，如果使用自定义的元类来创建签名对象也是很有意义的。下面的示例展示了这种替代方案是如何实现的：

```
from inspect import


Signature, Parameter

def


make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for


name in


names]
    return


Signature(parms)
```

```
class StructureMeta


(type):
    def


__new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields
        return


super().__new__(cls, clsname, bases, clsdict)

class Structure


(metaclass=StructureMeta):
    _fields = []
    def


__init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for


name, value in


bound_values.arguments.items():
            setattr(self, name, value)
# Example



class Stock


(Structure):
    _fields = ['name', 'shares', 'price']

class Point
```

```
(Structure):
    _fields = ['x', 'y']
```

当定义定制化的签名时，把签名对象保存到一个特殊的属性__signature__中常常是很有用的。如果这么做了，使用了inspect模块的代码在执行反射（introspection）操作时将能够获取到签名并将其作为函数的调用约定。示例如下：

```
>>> import inspect



>>> print


(inspect.signature(Stock))
(name, shares, price)
>>> print


(inspect.signature(Point))
(x, y)
>>>
```

9.17 在类中强制规定编码约定

9.17.1 问题

我们的程序由一个庞大的类继承体系组成，我们想强制规定一些编码约定（或者做一些诊断工作），使得维护这个程序的程序员能够轻松一些。

9.17.2 解决方案

如果想对类的定义进行监控，通常可以用元类来解决。一个基本的元类通常可以通过从type中继承，然后重定义它的__new__()或者__init__()方法即可。示例如下：

```
class MyMeta

(type):
    def

__new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined

        # bases is tuple of base classes
```

```
        # clsdict is class dictionary


        return

super().__new__(cls, clsname, bases, clsdict)
```

另一种方式是定义__init__():

```
class MyMeta


(type):
    def


__init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined



        # bases is tuple of base classes



        # clsdict is class dictionary
```

要使用元类，一般来说会将其作用到一个顶层基类上，然后让其他子类继承之。示例如下：

```
class Root


(metaclass=MyMeta):
    pass




class A


(Root):
    pass




class B


(Root):
    pass
```

元类的一个核心功能就是允许在定义类的时候对类本身的内容进行检查。在重新定义的__init__()方法中，我们可以自由地检查类字典、基类以及其他更多信息。此外，一旦为某个类指定了元类，该

类的所有子类都会自动继承这个特性。因此，聪明的框架实现者可以在庞大的类继承体系中为其中一个顶层基类指定一个元类，然后就可以获取到位于该基类之下的所有子类的定义了。

下面是一个有些异想天开的例子，这里的元类可用来拒绝类定义中包含大小写混用的方法名（也许这就是为了恶心一下Java程序员）：

```
class NoMixedCaseMeta


(type):
    def


__new__(cls, clsname, bases, clsdict):
        for


name in


clsdict:
            if


name.lower() != name:
                raise TypeError


('Bad attribute name: ' + name)
            return


super().__new__(cls, clsname, bases, clsdict)

class Root
```

```
(metaclass=NoMixedCaseMeta):
    pass



class A

(Root):
    def

foo_bar(self):     # Ok


        pass



class B

(Root):
    def

fooBar(self):     # TypeError


        pass
```

作为一个更加高级而且有用的例子，下面定义的元类可检查子类中是否有重新定义的方法，确保它们的调用签名和父类中原始的方法相同。

```
from inspect import


signature
import logging




class MatchSignaturesMeta


(type):
    def


__init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
        for


name, value in


clsdict.items():
            if


name.startswith('_') or not


callable(value):
                continue
```

```
            # Get the previous definition (if any) and compare th



            prev_dfn = getattr(sup,name,None)
            if



prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if



prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s
                                value.__qualname__, prev_sig, val
# Example



class Root



(metaclass=MatchSignaturesMeta):
    pass





class A



(Root):
    def



foo(self, x, y):
        pass





def
```

```
spam(self, x, *, z):
    pass



# Class with redefined methods, but slightly different signatures


class B

(A):
    def

foo(self, a, b):
        pass



    def

spam(self,x,z):
        pass
```

如果运行上述代码，将得到如下的输出：

```
WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (se
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self,
```

类似这样的告警信息可能对于捕获微妙的程序bug会很有帮助。比方说，某个方法依赖于传递给它的关键字参数，如果子类修改了参数名称那么就会崩溃。

9.17.3　讨论

在一个大型的面向对象程序中，有时候通过元类来控制类的定义会十分有用。元类可以监视类的定义，可用来警告程序员那些可能会被忽视的潜在问题（比如使用了不兼容的方法签名）。

有些人可能会认为像这种错误最好用程序分析工具或者IDE来捕获。确实，这类工具是非常有用的。但是，如果正在创建一个由其他人使用的框架或者库，那么通常是无法控制其他开发者的开发流程的（如果他们不用这类工具怎么办？）。因此，对于特定类型的应用，在元类中做一些额外的检查是很有意义的，这类检查常常使得产品有着更好的用户体验。

至于在元类中是重新定义__new__()还是__init__()，这取决于我们打算如何使用得到的结果类。__new__()会在类创建之前先得到调用，当元类想以某种方式修改类的定义时（通过修改类字典中的内容）一般会用这种方法。而__init__()方法会

在类已经创建完成之后才得到调用，如果想编写代码同完全成形（fully formed）的类对象打交道，那么重新定义__init__()会很有用。在最后那个示例中我们必须重新定义__init__()。因为这里用到了super()函数来查找父类中的定义，而这只有当类实例已经被创建出来且方法解析顺序（MRO）已经设定之后才行得通。

最后那个示例也展示了对Python函数签名对象的使用。从本质上说，元类首先获取类中的每一个可调用型的定义（函数、方法等），然后查找它们是否在基类中也有一个定义，如果有的话就通过inspect.signature()来比较它们的调用签名是否一致。

最后但同样重要的是，super(self, self)这行代码并不存在输入错误。当使用元类时，很重要的一点是要意识到self实际上是一个类对象。因此，这行代码实际上是用来寻找位于类层次结构中更高层次上的定义，它们组成了self的父类。

9.18　通过编程的方式来定义类

9.18.1　问题

我们编写的代码最终需要创建一个新的类对象。我们想到将组成类定义的源代码发送到一个字符串中，然后利用类似exec()这样的函数来执行，但是我们希望能有一个更加优雅的解决方案。

9.18.2　解决方案

我们可以使用函数types.new_class()来实例化新的类对象。所有要做的就是提供类的名称、父类名组成的元组、关键字参数以及一个用来产生类字典（class dictionary）的回调，类字典中包含着类的成员。示例如下：

```
# stock.py

# Example of making a class manually from parts

# Methods
```

```
def

__init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price

def

cost(self):
    return

self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}

# Make a class

import types

Stock = types.new_class('Stock', (), {}, lambda

ns: ns.update(cls_dict))
Stock.__module__ = __name__
```

这么做会产生一个普通的类对象，和所期望的

结果一样：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
4555.0
>>>
```

在调用完types.new_class()之后对Stock.__module__的赋值操作是这个解决方案中的微妙之处。每当定义一个类时，其__module__属性中包含的名称就是定义该类时所在的模块名。这个名称会用来为__repr__()这样的方法产生输出，同时也会被各种库所用，比如pickle。因此，为了让创建的类成为一个“正常”的类，需要保证将__module__属性设置妥当。

如果想创建的类还涉及一个不同的元类，可以在types.new_class()的第三个参数中进行指定。示例如下：

```
>>> import abc



>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMet
...                            lambda


ns: ns.update(cls_dict))
```

```
...
>>> Stock.__module__ = __name__
>>> Stock
<class

'__main__

.Stock'>
>>> type(Stock)
<class

'abc

.ABCMeta'>
>>>
```

第三个参数中还可以包含其他的关键字参数。例如，下面这个类定义：

```
class Spam

(Base, debug=True, typecheck=False):
    ...
```

转换成new_class()调用后是这样的：

```
Spam = types.new_class('Spam', (Base,),
                       {'debug': True, 'typecheck': False},
```

```
                    lambda

ns: ns.update(cls_dict))
```

new_class()的第四个参数是最为神秘的。但它实际上是一个接受映射型对象的函数，用来产生类的命名空间。这通常都会是一个字典，但实际上可以是任何由__prepare__()方法（见9.14节）返回的对象。这个函数应该使用update()方法或者其他的映射操作为命名空间中添加新的条目。

9.18.3 讨论

能够制造出新的类对象在某些特定的上下文中会很有用。其中一个我们比较熟悉的例子和collections.namedtuple()函数有关。示例如下：

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'p
>>> Stock
<class '__main__.Stock'>
>>>
```

和我们本节展示的技术不同，namedtuple()使用了exec()。但是，下面这个简单的函数可直接创建出类：

```
import operator

import types

import sys

def

named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors

    cls_dict = { name: property(operator.itemgetter(n))
                 for

n, name in

enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict

    def

__new__(cls, *args):
        if

len(args) != len(fieldnames):
            raise TypeError
```

```
('Expected {} arguments'.format(len(fieldnames)))
        return


tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class



    cls = types.new_class(classname, (tuple,), {},
                          lambda


ns: ns.update(cls_dict))

    # Set the module to that of the caller



    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return


cls
```

上述代码的最后部分利用了所谓的“frame hack”技巧，通过sys._getframe()来获取调用者所在的模块名称。有关frame hack的另一个例子可在2.15节中找到。

下面的示例展示了上述代码是如何工作的：

```
>>> Point = named_tuple('Point', ['x', 'y'])
```

```
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print


('%s %s' % p)
4 5
>>>
```

本节使用的技术中一个重要的方面在于对元类提供了适当支持。我们可能会倾向于通过直接实例化一个元类来创建类。示例如下：

```
Stock = type('Stock', (), cls_dict)
```

这种方法的问题在于它忽略了某些重要的步骤，比如调用元类的__prepare__()方法。通过采用types.new_class()，可以保证所有必要的初始化步骤都能得到执行。例如，在types.new_class()中给定的第四个参数是一个回调函数，它所接受的映射型对

象正是由__prepare__()方法返回的。

如果只想执行准备步骤，可以使用types.prepare_class()。示例如下：

```
import types

metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metac
```

这么做会找到合适的元类并调用它的__prepare__()方法。元类、剩下的关键字参数以及准备好的命名空间都会得到返回。

要获得更多信息，请参考 PEP 3115（http://www.python.org/dev/peps/pep-3115 ）以及Python的相关文档（http://docs.python.org/3/reference/datamodel.html%2
）。

9.19　在定义的时候初始化类成员

9.19.1　问题

我们想在定义类的时候对部分成员进行初始化，而不是在创建类实例的时候完成。

9.19.2　解决方案

在定义类的时候执行初始化或者配置操作是元类的经典用途。从本质上说，元类是在定义类的时候触发执行，此时可以执行额外的步骤。

下面的示例采用这种思想创建了一个类似于collections模块中命名元组的类：

```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(c
        return super().__new__(cls,args)
```

上述代码允许我们定义简单的基于元组的数据结构，示例如下：

```
class Stock(StructTuple):
    _fields = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields = ['x', 'y']
```

可以看看如何使用它们：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

9.19.3 讨论

本节中，类StructTupleMeta接受类属性_fields

中的属性名称，并将它们转换为属性方法，使得这些方法能够访问到元组的某个特定槽位。函数operator.itemgetter()创建了一个访问器函数（accessor function），而函数property()将其转换成一个property属性。

本节中最为棘手的部分在于如何知道不同的初始化步骤在什么时候发生。StructTupleMeta中的__init__()方法针对每个定义的类只会调用一次。参数cls代表着所定义的类。从本质上说，我们给出的代码利用类变量_fields来接受新定义的类，然后为其添加一些新的部分。

类StructTuple作为公共基类让用户从它继承。类中的__new__()方法负责产生新的实例。这里对__new__()的使用有些不同寻常，部分原因在于我们修改了元组的调用签名，这使得现在的调用约定看起来就和普通的调用方式一致了：

```
s = Stock('ACME', 50, 91.1)        # OK

s = Stock(('ACME', 50, 91.1))        # Error
```

和__init__()不同，__new__()方法会在类实例创建出来之前得到触发。由于元组是不可变对象（immutable），一旦它们被创建出来就无法再做任何修改了。因此，__init__()方法在类实例创建的过程中触发的时机太晚，以至于没法按我们想要的方式修改实例。这就是为什么我们要定义__new__()的原因。

尽管本节的内容比较短小，但通过仔细地学习和研究后，读者对于Python类的定义、类实例的创建过程以及元类和类中不同的特殊方法将在何时得到调用有着深刻的理解，这是大有益处的。

PEP 422（http://www.python.org/dev/peps/pep-0422 ）中还提供了一种可选的替代方案来完成本节中描述的任务。但是，在写作本书时，这份PEP还没有得到采纳和接受。尽管如此，如果你使用的Python版本要高于3.3的话还是值得去看一看的。

9.20　通过函数注解来实现方法重载

9.20.1　问题

我们已经学习过函数参数注解方面的知识，而我们想利用这种技术通过基于参数类型的方式来实现多分派（multiple-dispatch，或称为方法重载）。但是并不清楚这其中要涉及哪些技术，甚至对于这么做是否为一个好主意还存有疑虑。

9.20.2　解决方案

本节的思想基于一个简单的事实——即，由于Python允许对参数进行注解，那么如果可以像下面这样编写代码就好了：

```
class Spam

:
    def

bar(self, x:int, y:int):
        print
```

```
('Bar 1:', x, y)
    def


bar(self, s:str, n:int = 0):
        print


('Bar 2:', s, n)

s = Spam()
s.bar(2, 3)         # Prints Bar 1: 2 3



s.bar('hello')         # Prints Bar 2: hello 0
```

下面的解决方案正是应对于此，我们使用了元类以及描述符来实现：

```
# multiple.py



import inspect



import types




class MultiMethod
```

```
:
    '''

    Represents a single multimethod.

    '''

    def

__init__(self, name):
        self._methods = {}
        self.__name__ = name

    def

register(self, meth):
        '''

        Register a new method as a multimethod

        '''

        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations

        types = []
```

```
        for

name, parm in

sig.parameters.items():
            if

name == 'self':
                continue


            if

parm.annotation is

inspect.Parameter.empty:
                raise TypeError

(
                    'Argument {} must be annotated with a type'.f
                )
            if not

isinstance(parm.annotation, type):
                raise TypeError

(
                    'Argument {} annotation must be a type'.forma

                )
            if

parm.default is not
```

```
inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
            types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def


__call__(self, *args):
        '''



        Call a method based on type signature of the arguments



        '''



        types = tuple(type(arg) for


arg in


args[1:])
        meth = self._methods.get(types, None)
        if


meth:
            return


meth(*args)
        else


:
            raise TypeError
```

```
('No matching method for types {}'.format(types))

    def

__get__(self, instance, cls):
        '''

        Descriptor method needed to make calls work in a class

        '''

        if

instance is not

None:
            return

types.MethodType(self, instance)
        else

:
            return

self

class MultiDict

(dict):
    '''
```

```
    Special dictionary to build multimethods in a metaclass

    '''

    def

__setitem__(self, key, value):
        if

key in

self:
            # If key already exists, it must be a multimethod or



            current_value = self[key]
            if

isinstance(current_value, MultiMethod):
                current_value.register(value)
            else

:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else

:
            super().__setitem__(key, value)

class MultipleMeta
```

```
(type):
    '''

    Metaclass that allows multiple dispatch of methods

    '''

    def

__new__(cls, clsname, bases, clsdict):
        return

type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def

__prepare__(cls, clsname, bases):
        return

MultiDict()
```

要使用这个类，可以像这样编写代码：

```
class Spam
```

```
(metaclass=MultipleMeta):
    def


bar(self, x:int, y:int):
        print


('Bar 1:', x, y)
    def


bar(self, s:str, n:int = 0):
        print


('Bar 2:', s, n)

# Example: overloaded __init__



import time



class Date


(metaclass=MultipleMeta):
    def


__init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def


__init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)
```

下面的交互式会话可验证我们的代码是否能按预期工作：

```
>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 42, in __call__
    raise TypeError


('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 's

>>> # Overloaded __init__



>>> d = Date(2012, 12, 21)
>>> # Get today's date



>>> e = Date()
>>> e.year
2012
>>> e.month
12
>>> e.day
3
>>>
```

9.20.3 讨论

老实说，本节中出现了大量的“魔法”才使得这个方案能适用于现实环境中的代码。但是，这个方案深入挖掘了元类和描述符的内部工作原理，并强化了其中的一些概念。因此，就算可能不会直接应用本节中的方案，但其中的一些思想可能会影响到其他涉及元类、描述符和函数注解方面的编程技术。

相对来说，上述实现中的主要思想是比较简单的。元类MutipleMeta使用__prepare__()方法来提供一个定制化的类字典，将其作为MultiDict的一个类实例。与普通的字典不同，当设定字典中的条目时，MultiDict会检查条目是否已经存在。如果已经存在，则重复的条目会被合并到MultiMethod的一个类实例中去。

MultiMethod的类实例会通过构建一个从类型签名到函数的映射关系来将方法收集到一起。在构建的时候，我们通过函数注解来收集这些签名并构建出映射关系。这些都是在MultiMethod.register()方法中完成的。关于这个映射，一个至关重要的地方在于为了实现多方法重载，因此必须给所有的参数都指定类型，否则就会出错。

为了让MultiMethod的类实例能够表现为一个可调用对象，我们实现了__call__()方法。该方法通过所有的参数（除了 self 之外）构建出一个类型元组，然后在内部的映射关系中找到对应的方法并调用它。实现__get__()方法是为了让MultiMethod的类实例能够在类定义中正常工作。在我们给出的实现中，__get__()方法被用来创建合适的绑定方法。示例如下：

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

诚然，本节中虽然涉及多项编程技术，但不幸的是我们还需要考虑一下其中存在的局限性。第一，这个解决方案中不能使用关键字参数。示例如下：

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'
```

```
>>> s.bar(s='hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

也许可以通过某种方式增加对关键字参数的支持，但这就需要一种完全不同的方法来实现方法映射了。问题的根源在于关键字参数不是以某种特定顺序出现的。当和位置参数混在一起时，我们很快会得到一堆杂乱排列的参数，迫使我们不得不在__call__()方法中以某种方式进行整理。

本节给出的方案对于继承的支持也非常有限。例如，下面的代码是无法工作的：

```
class A

:
    pass



class B

(A):
    pass
```

```
class C

:
    pass



class Spam

(metaclass=MultipleMeta):
    def

foo(self, x:A):
        print

('Foo 1:', x)

    def

foo(self, x:C):
        print

('Foo 2:', x)
```

失败的原因在于注解x:A无法匹配到子类的实例上（比如B的实例）。示例如下：

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
```

```
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 44, in __call__
    raise TypeError


('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

除了使用元类和函数注解之外，还可以通过装饰器来实现类似的功能。示例如下：

```
import types



class multimethod


:
    def


__init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def


match(self, *types):
```

```
        def

register(func):
            ndefaults = len(func.__defaults__) if

func.__defaults__ else

0
            for

n in

range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return

self
        return

register

    def

__call__(self, *args):
        types = tuple(type(arg) for

arg in

args[1:])
        meth = self._methods.get(types, None)
        if

meth:
            return
```

```
meth(*args)
        else

:
            return

self._default(*args)

    def

__get__(self, instance, cls):
        if

instance is not

None:
            return

types.MethodType(self, instance)
        else

:
            return

self
```

要使用装饰器的版本，可以像这样编写代码：

```
class Spam
```

```
:
    @multimethod
    def


bar(self, *args):
        # Default method called if no match



        raise TypeError


('No matching method for bar')

    @bar.match(int, int)
    def


bar(self, x, y):
        print


('Bar 1:', x, y)

    @bar.match(str, int)
    def


bar(self, s, n = 0):
        print


('Bar 2:', s, n)
```

采用装饰器的解决方案和前面的实现方案有着相同的局限性（即，不支持关键字参数，对继承的

支持不佳）。

如果不出什么意外，最好还是不要在通用的代码中使用多分派。在一些特殊情况下这或许会是有意义的，比如某个程序需要根据某种形式的模式匹配来分派不同的方法。例如，在8.21节中描述过的访问者模式也许可以改写到一个类中，通过某种方式来使用多方法分派。但是除此之外，选择更加简单的方案通常都绝不会是个坏主意（不同的方法使用不同的名称即可）。

考虑通过不同的方式来实现多分派的思想已经在Python用户社区中存在多年了。对于这个主题的讨论，请参见Python之父Guido van Rossum发表的一篇博文“Five-Minute Multimethods in Python”（http://www.artima.com/weblogs/viewpost.jsp?thread=101605 ）。

9.21 避免出现重复的属性方法

9.21.1 问题

我们正在编写一个类，而我们不得不重复定义一些执行了相同任务的属性方法，比如说做类型检查。我们想简化代码，解决代码重复的问题。

9.21.2 解决方案

考虑下面这个简单的类，这里的属性都用property方法进行了包装：

```
class Person



:
    def



__init__(self, name ,age):
        self.name = name
        self.age = age

    @property
    def



name(self):
        return
```

```
self._name

    @name.setter
    def


name(self, value):
        if not


isinstance(value, str):
            raise TypeError


('name must be a string')
        self._name = value

    @property
    def


age(self):
        return


self._age

    @age.setter
    def


age(self, value):
        if not


isinstance(value, int):
            raise TypeError


('age must be an int')
        self._age = value
```

可以看到，我们编写的很多代码仅仅只是强制对属性值做类型断言。每当看到自己的代码变成这个样子时，应该考虑通过各种不同的方法来简化代码。一种可能的方式就是创建一个函数，让它为我们定义这个属性并返回给我们。示例如下：

```
def

typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def

prop(self):
        return

getattr(self, storage_name)

    @prop.setter
    def

prop(self, value):
        if not

isinstance(value, expected_type):
            raise TypeError

('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)
    return
```

```
prop

# Example use



class Person


:
    name = typed_property('name', str)
    age = typed_property('age', int)
    def


__init__(self, name, age):
        self.name = name
        self.age = age
```

9.21.3　讨论

本节说明了内层函数或者闭包的一个重要特性——即，用它们编写出的代码工作起来很像宏。示例中的函数typed_property()可能看起来有点怪，但它实际上只是在为我们生成属性代码，并返回产生的属性对象。因此，当在类中使用它时就好像把出现在typed_property()中的代码放置到了类定义中一样。尽管getter和setter属性方法访问的是局部变量，比如name、expected_type和storage_name，这也没问题——那些值都保存在闭包中了。

如果使用函数functools.partial()，还可以让本节

中的示例变得更加有趣。例如，可以这么做：

```
from functools import


partial

String = partial(typed_property, expected_type=str)
Integer = partial(typed_property, expected_type=int)
# Example:



class Person


:
    name = String('name')
    age = Integer('age')
    def


__init__(self, name, age):
        self.name = name
        self.age = age
```

这份代码看起来就很像我们在8.13节中展示的一些类型系统描述符的代码了。

9.22　以简单的方式定义上下文管理器

9.22.1　问题

我们想实现新形式的上下文管理器，然后在with语句中使用。

9.22.2　解决方案

编写一个新的上下文管理器，其中最直接的一种方式就是使用contextlib模块中的@contextmanager装饰器。在下面的示例中，我们用上下文管理器来计时代码块的执行时间：

```
import time

from contextlib import

contextmanager

@contextmanager
def

timethis(label):
```

```
    start = time.time()
    try

:
        yield

    finally

:
        end = time.time()
        print

('{}: {}'.format(label, end - start))

# Example use

with

timethis('counting'):
    n = 10000000
    while

n > 0:
        n -= 1
```

在timethis()函数中，所有位于yield之前的代码会作为上下文管理器的__enter__()方法来执行。而所有位于yield之后的代码会作为__exit__()方法执行。如果有异常产生，则会在yield语句中抛出。

下面是一个更加高级的上下文管理器，其中实现了对列表对象的处理：

```
@contextmanager

def list_transaction(orig_list):

    working = list(orig_list)

    yield working

    orig_list[:] = working
```

这里采用的思路就是只有当整个代码块执行结束且没有产生任何异常时，此时对列表做出的修改才会真正生效。下面用一个例子来说明：

```
>>> items = [1, 2, 3]

>>> with list_transaction(items) as working:
```

```
...
     working.append(4)
...
     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...
     working.append(6)
...
     working.append(7)
```

```
...
     raise RuntimeError('oops')
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>
```

9.22.3 讨论

一般来说，要编写一个上下文管理器，需要定

义一个带有__enter__()和__exit__()方法的类，就像下面这样：

```
import time

class timethis:
    def __init__(self, label):
        self.label = label
    def __enter__(self):
        self.start = time.time()
    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{}: {}'.format(self.label, end - self.start))
```

虽然这么做也并非很难，但是比起直接使用@contextmanager还是要繁琐许多。

@contextmanager只适用于编写自给自足型（self-contained）的上下文管理器函数。如果有一些对象（比如文件、网络连接或者锁）需要支持在with语句中使用，那么还是需要分别实现__enter__()和__exit__()方法。

9.23　执行带有局部副作用的代码

9.23.1　问题

我们正在使用exec()在调用方的作用域下执行一段代码，但是当执行结束后，得到的结果似乎在当前作用域下是不可见的。

9.23.2　解决方案

为了更好地理解这个问题，我们做一个小小的实验。首先，我们在全局命名空间下执行一段代码：

```
>>> a = 13

>>> exec('b = a + 1')

>>> print(b)

14

>>>
```

现在，让我们在一个函数内部再次做同样的实验：

```
>>> def test():
...
    a=13
...
    exec('b = a + 1')
...
    print(b)
...
>>> test()
```

```
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

  File "<stdin>", line 4, in test

NameError: global name 'b' is not defined

>>>
```

可以看到，我们遇到了NameError异常，就好像exec()语句从未实际执行过一样。如果打算将exec()的执行结果用在稍后的计算中，那么这就成了问题。

要解决这类问题，需要使用locals()函数在调用exec()之前获取一个保存了局部变量的字典。紧接着，就可以从本地字典中提取出修改过的值。示例如下：

```
>>> def
```

```
test():
...


     a = 13
...


     loc = locals()
...


     exec


('b = a + 1')
...


     b = loc['b']
...     print


(b)
...



>>> test()
14
>>>
```

9.23.3 讨论

在实践中要正确使用exec()其实是非常具有技巧性的。事实上，在大多数考虑使用exec()的情况中，可能存在着更加优雅的解决方案（例如装饰

器、闭包、元类等）。

但是如果仍然必须使用exec()，本节列出了一些正确使用它的原则。默认情况下，exec()是在调用方的局部和全局作用域中执行代码的。然而在函数内部，传递给exec()的局部作用域是一个字典，而这个字典是实际局部变量的一份拷贝。因此，如果在exec()中执行的代码对局部变量做出了任何修改，这个修改绝不会反映到实际的局部变量中去。下面我们用另一个例子来演示这个效果：

```
>>> def


test1():
...


    x = 0
...


    exec


('x += 1')
...


    print


(x)
...
```

```
>>> test1()
0
>>>
```

正如解决方案中展示的那样，当调用locals()来获取局部变量时，传递给exec()的是局部变量的拷贝。而在exec()执行完毕之后，通过检查字典中的值，就能获取到修改过的变量值。下面的实验可验证这一点：

```
>>> def

test2():
...

     x = 0
...

     loc = locals()
...

     print

('before:', loc)
...

     exec

('x += 1')
...
```

```
    print

('after:', loc)
...

    print

('x =', x)
...



>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

仔细观察最后一步的输出。除非从loc中将修改过的值写回x，否则变量x会保持不变。

每当使用locals()时都需要小心操作的顺序问题。每次调用它时，locals()将会接受局部变量的当前值，然后覆盖字典中的对应条目。观察下面这个实验的结果：

```
>>> def

test3():
...
```

```
     x = 0
...
     loc = locals()
...
     print
(loc)
...
     exec
('x += 1')
...
     print
(loc)
...
     locals()
...
     print
(loc)
...

>>> test3()
{'x': 0}
```

```
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>
```

注意最后对locals()的调用是如何导致x被覆盖的。

除了使用locals()之外，另一种可选的方式是自己创建字典并传递给exec()。示例如下：

```
>>> def


test4():
...


    a = 13
...


    loc = { 'a' : a }
...


    glb = { }
...


    exec


('b = a + 1', glb, loc)
...


    b = loc['b']
```

```
...

    print

(b)
...



>>> test4()
14
>>>
```

对于大部分针对exec()的应用，这可能就是优秀的实践方式了。我们需要确保exec()中访问的变量在全局和局部字典中经过恰当的初始化。

最后但同样重要的是，在使用exec()之前，应该问问自己是否还有其他可选的方案。许多可能会考虑使用 exec()的问题都可以用闭包、装饰器、元类或者其他元编程的特性来替代。

9.24 解析并分析Python源代码

9.24.1 问题

我们想编写程序来解析Python源代码并对此进行一些分析工作。

9.24.2 解决方案

大部分程序员都知道Python可以执行以字符串形式提供的源代码。示例如下：

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec


('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>
```

但是，我们可以使用ast模块将Python源代码编译为一个抽象语法树（AST），这样就可以分析源代码了。示例如下：

```
>>> import ast



>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None))], orelse=[])])"
>>>
```

对语法树的分析需要读者自己做一些研究，但总的来说，语法树是由一些AST节点组成的。同这些节点打交道的最简单的方式就是定义一个访问者类，在类中实现各种visit_NodeName()方法，这里的NodeName可以匹配到所感兴趣的节点上来。下面的示例给出了这样一个类，它可以记录那些被加

载、保存和删除过的名称信息。

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()
    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
for i in range(10):
    print(i)
del i
'''
    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

如果运行这个程序，会得到如下的输出：

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
```

```
Deleted: {'i'}
```

最后，AST节点可以通过compile()函数进行编译并执行。示例如下：

```
>>> exec(compile(top,'<stdin>', 'exec'))

0

1

2

3

4

5

6

7
```

```
8

9

>>>
```

9.24.3 讨论

由于可以分析源代码并从中得到有用的信息，这一事实可以让我们开始编写各种各样的代码分析、代码优化或者验证工具。例如，与其盲目地将一些代码片段传递给 exec()这样的函数，不如先将代码转换为一棵AST树，然后检查其中的一些细节来看看代码要完成哪些任务。也可以编写工具来检查模块的整份源码，并在此之上进行一些静态分析。

应该要提到的是，如果确实知道自己要做什么，那么也可以重写AST来表示新的源码。下面给出了一个装饰器的示例，可以降低函数体中可被全局访问的名称的数量。这是通过重新解析函数体的源码并重写AST，然后再重新创建函数的源码对象

来实现的。

```
# namelower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
        code += '\n'.join("{0} = __globals['{0}']".format(name)
                          for name in self.lowered_names)
        code_ast = ast.parse(code, mode='exec')

        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ','\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
        cl.visit(top)
```

```
        # Execute the modified AST
        temp = {}
        exec(compile(top,'','exec'), temp, temp)

        # Pull out the modified code object
        func.__code__ = temp[func.__name__].__code__
        return func
    return lower
```

要使用上述代码，可以编写如下形式的代码：

```
INCR = 1

@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR
```

这样，我们的装饰器就会将函数countdown()的源码改写为如下的形式：

```
def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR
```

在性能测试中，这使得该函数的运行速度快了大约20%。

现在，是否应该将这个装饰器作用到所有的函数上呢？很可能不是如此。但是，这个例子对一些非常高级的技术做了很好的说明。我们可以通过操纵AST、源代码以及其他一些技术来实现这些高级特性。

本节的灵感源自 ActiveState 上一个类似的例子（http://code.activestate.com/recipes/277940-decorator-for-bindingconstants-at-compile-time/ ），在那个例子中我们操纵了Python的字节码。用AST来实现则是一种层次更高的方法，可能会更直接一些。有关字节码方面的更多内容可参考下一节。

9.25 将Python源码分解为字节码

9.25.1 问题

我们想将Python源码分解为解释器所使用的底层字节码，以此了解代码在底层的详细细节。

9.25.2 解决方案

dis模块可用来将任何Python函数分解为字节码序列。示例如下：

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
  2           0 SETUP_LOOP              39 (to 42)
        >>    3 LOAD_FAST                0 (n)
              6 LOAD_CONST               1 (0)
              9 COMPARE_OP               4 (>)
             12 POP_JUMP_IF_FALSE       41

  3          15 LOAD_GLOBAL              0 (print)
             18 LOAD_CONST               2 ('T-minus')
             21 LOAD_FAST                0 (n)
             24 CALL_FUNCTION            2 (2 positional, 0 keywor
             27 POP_TOP

  4         28 LOAD_FAST                 0 (n)
```

```
             31 LOAD_CONST               3 (1)
             34 INPLACE_SUBTRACT
             35 STORE_FAST               0 (n)
             38 JUMP_ABSOLUTE            3
        >>   41 POP_BLOCK

  5     >>   42 LOAD_GLOBAL              0 (print)
             45 LOAD_CONST               4 ('Blastoff!')
             48 CALL_FUNCTION            1 (1 positional, 0 keyword
             51 POP_TOP                  0 (None)
             52 LOAD_CONST
             55 RETURN_VALUE
>>>
```

9.25.3 讨论

如果需要在非常底层的层次下研究程序的行为，那么dis模块会非常有帮助（例如，如果打算了解一些性能方面的特点时）。

由函数dis()所翻译出的原始字节码序列是这样的：

```
>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00Wt\x00\x00d\x04
\x01\x00\x01d\x00\x00S"
>>>
```

如果想自行解释这段代码，需要使用定义在opcode模块中的一些常量。示例如下：

```
>>> c = countdown.__code__.co_code
>>> import opcode



>>> opcode.opname[c[0]]
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[3]]
'LOAD_FAST'
>>>
```

讽刺的是，dis模块中居然没有任何函数能够让我们以可编程的方式来轻松处理这些字节码。下面这个生成器函数会接受原始的字节码序列，并将其转换为对应的操作代码和参数。

```
import opcode




def



generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while



i < n:
        op = codebytes[i]
        i += 1
        if
```

```
op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_
            extended_arg = 0
            i += 2
            if


op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue



        else


:
            oparg = None
        yield


(op, oparg)
```

要使用这个函数，可以像这样操作：

```
>>> for


op, oparg in


generate_opcodes(countdown.__code__.co_code):
...


     print
```

```
(op, opcode.opname[op], oparg)
...



120 SETUP_LOOP 39
124 LOAD_FAST 0
100 LOAD_CONST 1
107 COMPARE_OP 4
114 POP_JUMP_IF_FALSE 41
116 LOAD_GLOBAL 0
100 LOAD_CONST 2
124 LOAD_FAST 0
131 CALL_FUNCTION 2
1 POP_TOP None
124 LOAD_FAST 0
100 LOAD_CONST 3
56 INPLACE_SUBTRACT None
125 STORE_FAST 0
113 JUMP_ABSOLUTE 3
87 POP_BLOCK None
116 LOAD_GLOBAL 0
100 LOAD_CONST 4
131 CALL_FUNCTION 1
1 POP_TOP None
100 LOAD_CONST 0
83 RETURN_VALUE None
>>>
```

下面介绍一个鲜为人知的小技巧，我们可以将任何函数中感兴趣的原始字节码替换掉。这需要多做一些工作才能实现，下面给出的示例展示了其中需要涉及的技巧：

```
>>> def
```

```
add(x, y):
...

     return

x + y
...

>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code

>>> import types

>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...

     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_
...

     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...

     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault
```

使用这么花哨的技巧是很有可能将解释器弄崩溃的。但是，对于那些需要做高级优化和开发元编程工具的开发者来说，他们可能会真地倾向于去重新改写字节码。本节最后这个例子展示了如何去做。读者可以在 ActiveState 上看到另一个实际应用中的例子（http://code.activestate.com/recipes/277940-decorator-for-buildingconstants-at-cimpile- time ）。

第**10**章　模块和包

模块和包是任何大型项目的核心，就连Python安装程序本身也是一个包。本章的重点涉及有关模块和包的常见编程技术，例如如何组织包、将大型的模块分解成多个文件以及创建命名空间包（namespace package）。此外，本章也提到了关于自定义import语句行为的操作。

10.1　把模块按层次结构组织成包

10.1.1　问题

我们想把代码按照一定的层次结构组织成包。

10.1.2　解决方案

创建一个软件包结构是很简单的。只要把代码按照所希望的方式在文件系统上进行组织，并确保每个目录中都定义了一个*__init__.py*文件即可。例如：

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

一旦完成后，就可以执行各种各样的import语句了，比如：

```
import graphics.primitive.line


from graphics.primitive import

line
import graphics.formats.jpg as jpg
```

10.1.3 讨论

定义一个具有层次结构的模块就如同在文件系统上创建目录结构一样简单。*__init__.py* 文件的目的就是包含可选的初始化代码，当遇到软件包中不同层次的模块时会触发运行。比如，如果写下import graphics语句，文件*graphics/__init__.py* 会被导入并形成graphics命名空间中的内容。对于import graphics.formats.jpg这样的导入语句，文件*graphic/__ init__.py* 和*graphics/formats/__init__.py* 都会在最终导入文件*graphics/formats/jpg.py* 之前优先得到导入。

在大部分情况下，把*__init__.py* 文件留空也是可以的。但是，在某些特定的情况下*__init__.py* 文件中是需要包含代码的。例如，可以用*__init__.py*

文件来自动加载子模块，示例如下：

```
# graphics/formats/__init__.py



from . import


jpg
from . import


png
```

有了这样一个文件，用户只需要使用一条单独的import graphics.formats语句就可以导入jpg和png模块了，不需要再去分别导入graphics.formats.jpg和graphics.formats.png。

其他关于__*init*__*.py* 文件的常见用法包括从多个文件中把定义统一到一个单独的逻辑命名空间中，这有时候会在分解模块时用到。我们在10.4节中会讨论分解模块的问题。

一些精明的程序员会注意到在Python 3.3中就算不存在__*init*__*.py* 文件似乎也可以执行包的导入操作。如果不定义__*init*__*.py*，那么实际上是创建了一个称之为“命名空间包”（namespace package）

的东西，我们会在10.5节讨论这个主题。如果刚开始创建一个新的包，那么做法都是一样的，包括 *__init__.py* 文件也是一样。

10.2 对所有符号的导入进行精确控制

10.2.1 问题

当用户使用from module import *语句时，我们希望对从模块或包中导入的符号进行精确控制。

10.2.2 解决方案

在模块中定义一个变量__all__，用来显式列出可导出的符号名。示例如下：

```
# somemodule.py

def

spam():
    pass

def
```

```
grok():
    pass



blah = 42

# Only export 'spam' and 'grok'


__all__ = ['spam', 'grok']
```

10.2.3 讨论

尽管我们强烈反对使用from module import *这样的导入语句，但是在定义了大量符号的模块中还是能常看到这种用法。如果对此无动于衷的话，这种形式的导入会把所有不以下划线开头的符号名全部导出。换句话说，如果定义了__all__，那么只有显式列出的符号名才会被导出。

如果将__all__定义成一个空的列表，那么任何符号都不会被导出。如果__all__中包含有未定义的名称，那么在执行import语句时会产生一个AttributeError异常。

10.3　用相对名称来导入包中的子模块

10.3.1　问题

我们将代码组织成了一个包，想从其中一个子模块中导入另一个子模块，但是又不希望在import语句中硬编码包的名称。

10.3.2　解决方案

要在软件包的子模块中导入同一个包中其他的子模块，请使用相对名称来导入。例如，假设有一个名为mypackage的包，它在文件系统上组织成如下的形式：

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

如果模块mypackage.A.spam希望导入位于同一个目录中的模块grok，那么它应该包含一条这样的import语句：

```
# mypackage/A/spam.py

from . import

grok
```

如果模块mypackage.A.spam希望导入位于不同目录中的模块B.bar，可以使用下面的import语句来完成：

```
# mypackage/A/spam.py

from ..B import

bar
```

上面这两条import语句都是相对于spam.py文件的位置来进行操作的，而且其中没有包含最顶层包

的名称。

10.3.3 讨论

在包的内部，要在其中一个子模块中导入同一个包中其他的子模块，既可以通过给出完整的绝对名称，也可以通过上面示例中采用的相对名称来完成导入。示例如下：

```
# mypackage/A/spam.py

from mypackage.A import

grok        # OK

from . import

grok                 # OK

import grok

# Error (not found)
```

使用绝对名称的缺点在于这么做会将最顶层的包名称硬编码到源代码中，这使得代码更加脆弱，如果想重新组织一下结构会比较困难。例如，如果修改了包的名称，将不得不搜索所有的源代码文件并修改硬编码的名称。类似地，硬编码名称使得其他人很难移动这部分代码。例如，也许有人想安装两个不同版本的包，只通过名字来区分它们。如果采用相对名称导入，那么不会有任何问题，但是采用绝对名称导入则会使程序崩溃。

import语句中的.和..语法可能看起来比较有趣，把它们想象成指定目录名即可。.意味着在当前目录中查找，而..B表示在../*B* 目录中查找。这种语法只能用在from xx import yy这样的导入语句中。示例如下：

```
from . import

grok        # OK


import

.grok             # ERROR
```

尽管看起来似乎可以利用相对导入来访问整个文件系统，但实际上是不允许跳出定义包的那个目录的。也就是说，利用句点的组合形式进入一个不是Python包的目录会使得导入出现错误。

最后，应该要提到的是相对导入只在特定的条件下才起作用，即，模块必须位于一个合适的包中才可以。特别是，位于脚本顶层目录的模块不能使用相对导入。此外，如果包的某个部分是直接以脚本的形式执行的，这种情况下也不能使用相对导入。例如：

```
% python3 mypackage/A/spam.py     # Relative imports fail
```

另一方面，如果使用-m选项来执行上面的脚本，那么相对导入就可以正常工作了。示例如下：

```
% python3 -m mypackage.A.spam     # Relative imports work
```

有关包的相对导入的更多背景知识，请参阅PEP 328（http://www.python.org/dev/peps/ pep-0328）。

10.4 将模块分解成多个文件

10.4.1 问题

我们想将一个模块分解成多个文件。但是，我们不想破坏现在已经在使用这个模块的代码，而是希望可以将多个单独的文件在逻辑上统一成一个单独的模块。

10.4.2 解决方案

可以通过将模块转换为包的方式将模块分解成多个单独的文件。考虑下面这个简单的模块：

```
# mymodule.py



class A

:
    def

spam(self):
        print

('A.spam')
```

```
class B

(A):
    def

bar(self):
        print

('B.bar')
```

假设想将*mymodule.py* 分解为两个文件，每个文件中包含一个类的定义。要做到这点，可以从把*mymodule.py* 替换成目录*mymodule* 开始。在这个新的目录中创建如下的文件：

```
mymodule/
    __init__.py
    a.py
    b.py
```

在文件*a.py* 中填入下面的代码：

```
# a.py

class A
```

```
:
    def

spam(self):
        print

('A.spam')
```

而在文件*b.py* 中填入下面的代码:

```
# b.py


from .a import

A
class B

(A):
    def

bar(self):
        print

('B.bar')
```

最后在文件*__init__.py* 中将这两个文件绑定在一起：

```
# __init__.py



from .a import


A
from .b import


B
```

如果遵循以上的步骤，那么现在mypackage包在逻辑上就成为了一个单独的模块：

```
>>> import mymodule



>>> a = mymodule.A()
>>> a.spam()
A.spam
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

10.4.3　讨论

本节主要考虑的是一个设计上的问题。即，我们希望用户使用大量的小型模块，还是希望他们只使用一个单独的模块。例如，在一个庞大的代码库中，我们可以把所有的东西都分解成单独的文件，并让用户写下大量的import语句，就像下面这样：

```
from mymodule.a import

A
from mymodule.b import

B
...
```

这么做行得通，但是也给用户带来了很大的负担，因为他们需要知道不同的组件都存放在哪个文件中。通常，更加简单的方式是将事情统一起来，只用一条单独的import语句即可：

```
from mymodule import

A, B
```

对于这后一种情况，通常可以把mymodule想象成一个大型的源文件。但是，本节为大家演示了

如何在逻辑上把多个文件拼接成一个单独的命名空间的技术。关键之处在于创建一个包目录，并通过 *__init__.py* 文件将各个部分粘合在一起。

当分解模块时，需要对跨文件名的引用多加小心。例如，在本节示例中，class B需要把class A当做基类来访问。我们采用from .a import A这种相对于包的导入方式来获取class A的定义。

本节全篇都在使用相对于包的导入方式，避免在源代码中硬编码顶层模块名。这么做使得修改模块名或者将模块代码移动到别处都变得更加容易了（参见10.3节）。

可以对本节提到的技术进行扩展，引入“惰性”导入的概念。由前面的示例可知，*__init__.py* 文件一次性将所有需要的组件都导入进来。但是，对于非常庞大的模块，也许只希望在实际需要的时候才加载那些组件。为了实现这个目的，下面对 *__init__.py* 文件做了些微修改：

```
# __init__.py

def
```

```
A():
    from .a import


A
    return


A()

def


B():
    from .b import


B
    return


B()
```

在这个版本中，class A和class B已经由函数取代了，当首次访问它们时会加载所需的类。对于用户来说这不会有太大差别。示例如下：

```
>>> import mymodule



>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

惰性加载的主要缺点在于会破坏继承和类型检查机制。例如，我们可能需要稍微修改一下代码：

```
if
isinstance(x, mymodule.A):          # Error
    ...
if
isinstance(x, mymodule.a.A):          # Ok
    ...
```

有关惰性加载在真实世界中的应用，可以参考标准库中*multiprocessing/__init__.py* 中的源代码。

10.5 让各个目录下的代码在统一的命名空间下导入

10.5.1 问题

我们有一个庞大的代码库，其中有很多部分可能是由不同的人来维护和发布的。每个部分都组织成一个目录，就像包一样。但是，与其把每个部分都安装为单独命名的包，我们更想把所有的部分联合在一起，用一个统一的前缀来命名。

10.5.2 解决方案

基本上来说，这里的问题就是我们想定义一个顶层的Python包，把它作为命名空间来管理大量单独维护的子模块。这个问题常常会在大型的应用程序框架中出现，框架开发人员希望鼓励用户发布自己的插件或者附加的包。

要使各个单独的目录统一在一个公共的命名空间下，可以把代码像普通的Python包那样进行组织。但是对于打算合并在一起的组件，这些目录中的*__init__.py* 文件则需要忽略。为了说明这个过程，假设Python代码位于两个不同的目录中：

```
foo-package/
    spam/
        blah.py

bar-package/
    spam/
        grok.py
```

在这两个目录中，spam 用来作为公共的命名空间。注意到这两个目录中都没有出现__*init__.py*文件。

现在如果将 foo-package和bar-package 都添加到Python的模块查询路径中，然后尝试做一些导入操作，看看会发生什么：

```
>>> import sys



>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import spam.blah



>>> import spam.grok



>>>
```

将会注意到这两个不同的包目录魔法般地合并

在了一起，我们可以随意导入spam.blah或者spam.grok，不会遇到任何问题。

10.5.3 讨论

这里的工作原理用到了一种称之为“命名空间包”（namespace package）的特性。基本上来说，命名空间包是一种特殊的包，设计这种特性的意图就是用来合并不同目录下的代码，把它们放在统一的命名空间之下进行管理，就像示例中展示的那样。对于大型的框架而言，这种特性是很有帮助的。因为这样允许把框架的某些部分分解成单独安装的包。这样也使得人们可以轻松地制作第三方插件和针对框架的其他扩展。

创建命名空间包的关键之处在于确保在统一命名空间的顶层目录中不包含*__init__.py* 文件。当导入包的时候，这个缺失的*__init__.py* 文件会导致发生一些有趣的事情。解释器并不会因此而产生一个错误，相反，解释器开始创建一个列表，把所有恰好包含有这个包名的目录都囊括在内。此时就创建出了一个特殊的命名空间包模块，且在__path__变量中会保存一份只读形式的目录列表。示例如下：

```
>>> import spam
```

```
>>> spam.__path__
_NamespacePath(['foo-package/spam', 'bar-package/spam'])
>>>
```

__path__变量中保存的目录可用来进一步定位包中的子模块（例如，当导入 spam.grok或者spam.blah时）。

命名空间包的一个重要特性就是任何人都可以用自己的代码来扩展命名空间中的内容。例如，假设在自己的目录下添加了代码：

```
my-package/
    spam/
        custom.py
```

如果把自己的代码目录和其他的包一起添加到sys.path 中，那么就可以无缝地同其他的spam包合并在一起：

```
>>> import spam.custom



>>> import spam.grok



>>> import spam.blah
```

```
>>>
```

作为一种调试工具，想知道某个包是不是用来当做命名空间包的主要方式就是检查它的__file__属性。如果缺少这个属性，这个包就是命名空间。这也可以从包对象的字符串表示中看出来，如果是命名空间的话，其中会有“namespace”的字样。示例如下：

```
>>> spam.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
>>> spam
<module 'spam' (namespace)>
>>>
```

有关命名空间包的更多信息，可以在PEP 420（http://www.python.org/dev/peps/pep-0420 ）中找到。

10.6　重新加载模块

10.6.1　问题

因为对模块的源代码做了修改，我们想重新加载一个已经加载过了的模块。

10.6.2　解决方案

要重新加载一个之前已加载过的模块，可以使用imp.reload()来实现。示例如下：

```
>>> import spam



>>> import imp



>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

10.6.3　讨论

在开发和调试阶段，重新加载模块这一招常常

很有用。但是一般来说在生产环境中这么做是不安全的，因为它并不会总是按照期望的方式工作。

reload()操作会擦除模块底层字典（__dict__）的内容，并通过重新执行模块的源代码来刷新它。模块对象本身的标识并不会改变（即，调用id()的结果）。因此，这个操作会使得已经导入到程序中的模块得到更新。

但是，对于使用了from module import name这样的语句导入的定义，reload()是不会去更新的。为了说明其中的过程，考虑下面的代码：

```
# spam.py



def


bar():
    print


('bar')

def


grok():
    print


('grok')
```

现在开启一个新的交互式会话：

```
>>> import spam



>>> from spam import


grok
>>> spam.bar()
bar
>>> grok()
grok
>>>
```

不要退出Python，现在去编辑*spam.py* 的源代码，把grok()函数修改成下面这样：

```

def



grok():
    print



('New grok')
```

现在返回交互式会话执行reload()操作，并做以

下的试验：

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok()              # Notice old output
grok
>>> spam.grok()         # Notice new output
New grok
>>>
```

在这个例子中，将会发现有两个版本的grok()函数都被加载进来了。一般来说这不会是我们期望的结果，而且最终会变成让我们头疼的噩梦。

基于这个原因，在生产环境的代码中应该要避免重新加载模块。但是在调试或者在交互式会话中，当需要尝试一些新想法时这么做也未尝不可。

10.7　让目录或zip文件成为可运行的脚本

10.7.1　问题

我们的程序已经从一个简单的脚本进化为一个涉及多个文件的应用。我们希望能有某种简单的方法让用户来运行这个程序。

10.7.2　解决方案

如果应用程序已经进化为由多个文件组成的“庞然大物”，则可以把它们放在专属的目录中，并为之添加一个*__main__.py*文件。例如，可以创建一个这样的目录：

```
myapplication/
     spam.py
     bar.py
     grok.py
     __main__.py
```

如果有*__main__.py*，就可以在顶层目录中运行Python解释器，就像下面这样：

```
bash % python3 myapplication
```

解释器会把*__main__.py*文件作为主程序来执行。

这项技术在我们把所有的代码打包进一个zip文件中时同样有效。示例如下：

```
bash % ls
spam.py  bar.py  grok.py  __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

10.7.3 讨论

创建一个目录或zip文件，并在其中添加一个*__main__.py*，这是一种打包规模较大的Python应用程序的可行方法。但这和安装到Python标准库中的包有所不同，在这种情况下，代码并不是作为标准库中的模块来使用的。相反，这里只是把代码打包起来方便给其他人执行。

由于目录和zip文件同普通文件相比有一些小的区别，我们可能也想添加一个shell脚本来让执行步骤变得更加简单。例如，如果代码位于一个名

为*myapp.zip* 的文件中，则可以像下面这样创建一个顶层的脚本：

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

10.8　读取包中的数据文件

10.8.1　问题

我们的代码需要读取包中的一个数据文件，我们要尽可能的以可移植的方式来处理。

10.8.2　解决方案

假设包是按照下列方式组织的：

```
mypackage/
    __init__.py
    somedata.dat
    spam.py
```

现在假设文件spam.py要读取somedata.dat中的内容。要做到这点，可以使用下列代码来完成：

```
# spam.py

import pkgutil
```

```
data = pkgutil.get_data(__package__, 'somedata.dat')
```

得到的结果会保存在变量data中。这是一个字节串，其中包含了文件的原始内容。

10.8.3 讨论

要读取一个数据文件，我们可能会倾向于编写代码利用内建的I/O函数（比如open()）来完成。但是，这种方法存在几个问题。

首先，对于一个包来说，它无法控制解释器的当前工作目录。因此，任何I/O操作都必须使用文件名的绝对路径。由于每个模块都在__file__变量中保存了全路径，所以要获取文件的位置并非不可能，但是会很麻烦。

其次，包通常都会安装为.zip或者.egg文件，它们和文件系统中普通的目录保存文件的方式不同。因此如果尝试用open()打开包含在归档（archive）中的数据文件，这根本行不通。

pkgutil.get_data()函数是一种高级的工具，无论包以什么样的形式安装或安装到了哪里，都能够用它来获取数据文件。它能够完成工作并把文件内容

以字节串的形式返回给我们。

get_data()的第一个参数是包含有包名的字符串。我们可以直接提供这个字符串，或者使用__package__这个特殊变量。第二个参数是要获取的文件相对于包的名称。如果有必要，可以使用标准的UNIX路径名规则进入不同的目录中，只要最后的目录仍然在包的内部即可。

10.9 添加目录到sys.path中

10.9.1 问题

我们有一些Python代码无法导入，因为它们不在sys.path列出的目录中。我们想将新的目录添加到Python的路径里，但又不想将其硬编码到代码中。

10.9.2 解决方案

有两种常见的方法可以将新的目录添加到sys.path 中去。第一，可以通过使用PYTHONPATH环境变量来添加。示例如下：

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct 4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more informa
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

在一个用户程序中，这个环境变量可以在程序启动时或者通过某种形式的shell脚本来设定。

第二种方法是创建一个*.pth* 文件，然后像下面这样将目录列出来：

```
# myapplication.pth
/some/dir
/other/dir
```

这个*.pth* 文件需要放在Python的其中一个*site-packages* 目录中，一般来说位于*/usr/local/lib/python3.3/site-packages* 或者*~/.local/lib/python3.3/site-packages* 。在解释器启动的时候，只要*.pth* 文件中列出的目录存在于文件系统上，那么它们就会被添加到sys.path中。如果是要添加到整个系统级的Python解释器上，那么安装.pth文件可能需要管理员权限。

10.9.3　讨论

如果在确定文件的位置时遇到了麻烦，可能会倾向于编写代码来手动调整sys.path的值。示例如下：

```
import sys



sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

尽管这可以“工作”，但在实践中这种做法极度脆弱，应该尽可能避免这种做法。这种方法的部分问题在于将目录名称硬编码到了源码中。如果要将代码转移到一个新的位置时，这就会产生维护方面的问题了。通常更好的方法是在其他的地方对路径做配置，不用去直接编辑代码。

有时候，如果利用模块级的变量比如__file__来精心构建一个合适的绝对路径，也能够规避硬编码目录所带来的问题。示例如下：

```
import sys

from os.path import

abspath, join, dirname
sys.path.insert(0, abspath(dirname('__file__'), 'src'))
```

上面的代码将src目录添加到了sys.path中，而且src目录和执行插入操作的代码所在的目录是相同的。

目录*site-packages* 通常是第三方模块和包安装

的位置。如果代码也按照这种方式安装，那么这就是它们所处的位置。尽管用来配置路径的*.pth* 文件必须出现在*site-packages* 中，但其中记录的路径可以指向系统中任何希望的目录。因此，可以选择让代码保存在一个完全不同的目录中，只要这些目录都包含在*.pth* 文件中即可。

10.10 使用字符串中给定的名称来导入模块

10.10.1 问题

我们已经有了需要导入的模块名称，但是这个名称保存在一个字符串中。我们想在字符串上执行import命令。

10.10.2 解决方案

当模块或包的名称以字符串的形式给出时，可以使用importlib.import_module()函数来手动导入这个模块。示例如下：

```
>>> import importlib



>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

import_module基本上和import完成的步骤相同，但是import_module会把模块对象作为结果返回给你。我们只需要将它保存在一个变量里，之后把它当做普通的模块使用即可。

如果要同包打交道，import_module()也可以用来实现相对导入。但是，需要提供一个额外的参数。示例如下：

```
import importlib

# Same as 'from . import b'

b = importlib.import_module('.b', __package__)
```

10.10.3　讨论

采用import_module()手动导入模块的需求最常出现在当编写代码以某种方式来操作或包装模块时。例如，也许正在实现一个自定义的导入机制，需要通过模块的名称来完成加载并给加载进来的代码打上补丁。

在较老的代码中，有时候会看到用内建的__import__()函数来实现导入。尽管这样也行得通，但importlib.import_module()通常要更容易使用一些。

请参见10.11节中有关自定义导入过程的高级示例。

10.11　利用import钩子从远端机器上加载模块

10.11.1　问题

我们想对Python的import语句做定制化处理，实现以透明的方式从远端机器上加载模块。

10.11.2　解决方案

首先我们要对安全性方面的问题做严肃的免责声明。本节讨论的思路和技术如果缺少某种额外的安全和认证机制的保护将变得非常糟糕。也就是说，本节的主要目标实际上是对Python中import语句的内部工作原理做了深入的探讨。如果能消化本节的内容并理解内部的工作原理，那么将为此打下坚实的基础。今后面对定制化import的操作，无论是出于什么目的，我们都能应对自如。好了，让我们继续吧。

本节的核心目标是扩展import语句的功能。有好几种方法可以实现这个目标，但是为了说明起见，我们先把Python代码按照下列方式进行组织：

```
testcode/
    spam.py
    fib.py
    grok/
        __init__.py
        blah.py
```

这些文件的内容无关紧要，但是可以在每个文件中放一些简单的语句和函数，这样我们可以进行测试，当它们被导入时可以看到输出的结果。例如：

```
# spam.py


print


("I'm spam")

def


hello(name):
    print


('Hello %s' % name)

# fib.py


print


```

```
("I'm fib")

def


fib(n):
    if


n < 2:
        return


1
    else


:
        return


fib(n-1) + fib(n-2)

# grok/__init__.py



print


("I'm grok.__init__")

# grok/blah.py



print


("I'm grok.blah")
```

这么做的目的是允许这些文件能够以模块的形式从远端访问。也许最简单的方式就是在 Web 服务器上来发布这些模块。只需要进入 *testcode* 目录，然后像下面这样运行Python即可：

```
bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...
```

让服务器一直运行，然后启动一个新的Python解释器进程。确保可以通过 urllib 来访问这些远程文件。示例如下：

```
>>> from urllib.request import


urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print


(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

从服务器中加载源代码，这一思想将是本章余下内容的基础。具体来说就是，与其通过urlopen()函数手动从服务器上把源代码抓取下来，不如自定义import语句的行为，使其能够在幕后以透明的方式实现同样的目的。

第一种用来加载远程模块的方法就是创建一个显式的加载函数。示例如下：

```
import imp

import urllib.request

import sys

def

load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec

(code, mod.__dict__)
    return
```

```
mod
```

这个函数只是用来下载源代码的，利用compile()函数将其编译为code对象，然后在新创建的模块对象的字典中执行它。下面是使用这个函数的方法：

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:1
>>>
```

可以看到，对于简单的模块这么做是可行的。但是，这个功能并没有嵌入到常用的import语句中。而且如果要支持更加高级的组件，比如包，就需要扩展代码，这都需要做更多的工作才能实现。

更加高级的方法是创建一个自定义的导入器（importer）。实现这个目的的第一种方法是创建一个称之为元路径导入器（meta path importer）的

组件。示例如下：

```
# urlimport.py

import sys

import importlib.abc

import imp

from urllib.request import

urlopen
from urllib.error import

HTTPError, URLError
from html.parser import

HTMLParser

# Debugging

import logging

log = logging.getLogger(__name__)
```

```
# Get links from a given URL



def


_get_links(url):
    class LinkParser


(HTMLParser):
        def


handle_starttag(self, tag, attrs):
            if


tag == 'a':
                attrs = dict(attrs)
                links.add(attrs.get('href').rstrip('/'))
    links = set()
    try


:
        log.debug('Getting links from %s' % url)
        u = urlopen(url)
        parser = LinkParser()
        parser.feed(u.read().decode('utf-8'))
    except Exception as


e:
        log.debug('Could not get links. %s', e)
    log.debug('links: %r', links)
    return


links

class UrlMetaFinder
```

```
(importlib.abc.MetaPathFinder):
    def


__init__(self, baseurl):
        self._baseurl = baseurl
        self._links = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def


find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname,
        if


path is


None:
            baseurl = self._baseurl
        else


:
            if not


path[0].startswith(self._baseurl):
                return


None
            baseurl = path[0]

        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseur

        # Check link cache



        if
```

```
basename not in

self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Check if it's a package

        if

basename in

self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init_

            loader = UrlPackageLoader(fullurl)
            try

:
                loader.load_module(fullname)
                self._links[fullurl] = _get_links(fullurl)
                self._loaders[fullurl] = UrlModuleLoader(fullurl
                log.debug('find_module: package %r loaded', fulln
            except ImportError as

e:
                log.debug('find_module: package failed. %s', e)
                loader = None
            return

loader
```

```
        # A normal module


        filename = basename + '.py'
        if


filename in


self._links[baseurl]:
            log.debug('find_module: module %r found', fullname)
            return


self._loaders[baseurl]
        else


:
            log.debug('find_module: module %r not found', fullnar
            return


None

    def


invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links.clear()

# Module Loader for a URL



class UrlModuleLoader


(importlib.abc.SourceLoader):
    def
```

```
__init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def


module_repr(self, module):
        return


'<urlmodule %r from %r>' % (module.__name__, module.__file__)

    # Required method



    def


load_module(self, fullname):
        code = self.get_code(fullname)
        mod = sys.modules.setdefault(fullname, imp.new_module(ful
        mod.__file__ = self.get_filename(fullname)
        mod.__loader__ = self
        mod.__package__ = fullname.rpartition('.')[0]
        exec


(code, mod.__dict__)
        return


mod

    # Optional extensions



    def


get_code(self, fullname):
        src = self.get_source(fullname)
        return
```

```
compile(src, self.get_filename(fullname), 'exec')

    def


get_data(self, path):
        pass




    def


get_filename(self, fullname):
        return


self._baseurl + '/' + fullname.split('.')[-1] + '.py'

    def


get_source(self, fullname):
        filename = self.get_filename(fullname)
        log.debug('loader: reading %r', filename)
        if


filename in


self._source_cache:
            log.debug('loader: cached %r', filename)
            return


self._source_cache[filename]
        try


:
            u = urlopen(filename)
```

```
            source = u.read().decode('utf-8')
            log.debug('loader: %r loaded', filename)
            self._source_cache[filename] = source
            return


source
        except


(HTTPError, URLError) as


e:
            log.debug('loader: %r failed. %s', filename, e)
            raise ImportError


("Can't load %s" % filename)

    def


is_package(self, fullname):
        return


False

# Package loader for a URL



class UrlPackageLoader


(UrlModuleLoader):
    def


load_module(self, fullname):
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
        mod.__package__ = fullname
```

```
    def


get_filename(self, fullname):
        return


self._baseurl + '/' + '__init__.py'

    def


is_package(self, fullname):
        return


True

# Utility functions for installing/uninstalling the loader



_installed_meta_cache = { }
def


install_meta(address):
    if


address not in


_installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def


remove_meta(address):
    if
```

```
address in


_installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)
```

下面的交互式会话展示了应该如何使用上述代码:

```
>>> # importing currently fails



>>> import fib



Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Load the importer and retry (it works)



>>> import urlimport



>>> urlimport.install_meta('http://localhost:15000')
>>> import fib



I'm fib
>>> import spam
```

```
I'm spam
>>> import grok.blah


I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

在这个特定的解决方案中，我们把一个特殊的查询对象——UrlMetaFinder的实例安装到sys.meta_path的最后一个条目中。每当要导入模块时就会在sys.meta_path中查找对应的查询对象，以此来寻找模块。在这个示例中，如果在所有正常的位置上都找不到所需的模块，此时UrlMetaFinder实例就成了最后的救命稻草，会触发它来寻找所需的模块。

作为一般的实现方法，UrlMetaFinder类对用户指定的URL进行包装。在内部，查询器会通过给定的URL构建一组合法的链接。当出现导入的动作时，用模块名来同已知的链接进行对比。如果有匹配，此时就用UrlModuleLoader类来从远端机器上加载模块的源代码并创建出最终的模块对象作为结果。缓存链接的一个原因是为了避免对重复的导入做不必要的HTTP请求。

自定义导入功能的第二种方式是编写一个钩子（hook），直接将其插入到sys.path变量中，用来识别特定的目录命名模式。下面我们给*urlimport.py* 文件添加以下的类和支持函数：

```
# urlimport.py



# ... include previous code above ...



# Path finder class for a URL



class UrlPathFinder


(importlib.abc.PathEntryFinder):
    def


__init__(self, baseurl):
        self._links = None
        self._loader = UrlModuleLoader(baseurl)
        self._baseurl = baseurl

    def


find_loader(self, fullname):
        log.debug('find_loader: %r', fullname)
        parts = fullname.split('.')
        basename = parts[-1]
        # Check link cache
```

```
        if

self._links is

None:
            self._links = [] # See discussion


            self._links = _get_links(self._baseurl)

        # Check if it's a package


        if

basename in

self._links:
            log.debug('find_loader: trying package %r', fullname
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init_


            loader = UrlPackageLoader(fullurl)
            try

:
                loader.load_module(fullname)
                log.debug('find_loader: package %r loaded', fulln
            except ImportError as

e:
                log.debug('find_loader: %r is a namespace package
                loader = None
```

```
            return

(loader, [fullurl])

        # A normal module

        filename = basename + '.py'
        if

filename in

self._links:
            log.debug('find_loader: module %r found', fullname)
            return

(self._loader, [])
        else

:
            log.debug('find_loader: module %r not found', fullnar
            return

(None, [])

    def

invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links = None

# Check path to see if it looks like a URL

_url_path_cache = {}
def
```

```
handle_url(path):
    if


path.startswith(('http://', 'https://')):
        log.debug('Handle path? %s. [Yes]', path)
        if


path in


_url_path_cache:
            finder = _url_path_cache[path]
        else


:
            finder = UrlPathFinder(path)
            _url_path_cache[path] = finder
        return


finder
    else


:
        log.debug('Handle path? %s. [No]', path)

def


install_path_hook():
    sys.path_hooks.append(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Installing handle_url')

def


remove_path_hook():
    sys.path_hooks.remove(handle_url)
```

```
    sys.path_importer_cache.clear()
    log.debug('Removing handle_url')
```

要使用这个基于路径的查询器，只需要将URL添加到sys.path中。例如：

```
>>> # Initial import fails



>>> import fib



Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Install the path hook



>>> import urlimport



>>> urlimport.install_path_hook()

>>> # Imports still fail (not on path)



>>> import fib



Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
```

```
>>> # Add an entry to sys.path and watch it work
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

最后这个例子的关键在于handle_url()函数，我们将它添加到了sys.path_hooks中。当开始处理sys.path中的条目时，位于sys.path_hooks中的函数就被调用。如果这些函数中有任何一个返回了一个查询对象（finder object），就用这个查询对象来尝试为sys.path中的条目加载模块。

应该要指出的是，从远端导入的模块使用起来和其他的模块一样。示例如下：

```
>>> fib
```

```
<urlmodule 'fib' from

'http

://localhost

:15000/fib.py

'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect


>>> print

(inspect.getsource(fib))
# fib.py


print

("I'm fib")

def

fib(n):
    if

n < 2:
        return
```

```
1
    else

:
        return

fib(n-1) + fib(n-2)
>>>
```

10.11.3　讨论

在继续深入讨论本节中提到的技术之前，应该要强调的是Python的模块、包和导入机制是整个语言中最为复杂的部分之一——就算是最有经验的Python程序员对这部分的理解往往也不尽人意，除非他们愿意竭尽所能去挖掘底层的原理。有几份重要的文档值得去阅读，包括importlib模块的文档（http://docs.python.org/3/library/importlib.html ）以及PEP 302（http://www.python.org/dev/peps/pep-0302 ）。本书不会重复文档中的内容，但会讨论其中的一些要点。

首先，如果想创建一个新的模块对象（module object），可以使用imp.new_module()函数。示例如下：

```
>>> import imp
```

```
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

模块对象通常会有几个意料之中的属性，包括__file__（所加载模块的源文件名）和__package__（包的名称，如果有的话）。

其次，模块会被解释器做缓存处理。模块缓存可以在字典sys.modules中找到。由于存在缓存处理，通常我们就把缓存和模块的创建联合成一个单独的步骤来做。示例如下：

```
>>> import sys



>>> import imp



>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

这么做的主要原因是如果某个给定名称的模块已经存在的话，就会直接得到已经创建好的模块了。示例如下：

```
>>> import math



>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
>>>
```

由于创建模块是很简单的，所以可以直接编写简单的函数来处理，就像本节第一部分中的load_module()函数那样。这种方法的缺点是对于更加复杂的情况，处理会变得相当棘手，例如导入包的时候。为了能够处理包，将不得不重新实现大部分的底层逻辑，而这些东西已经是普通的import语句实现过了的（例如，检查目录、查找__init__.py文件、执行这些文件、建立起路径等）。

由于这种复杂性，因此通常更好的选择是直接扩展import语句的功能，而不是去定义自己的处理函数，这正是为何要这么做的原因之一。

扩展import语句是简单而直接的，但是其中涉及多个部件。从最高层级来看，import操作要处理一系列“元路径”查询器，这些查询器可以在sys.meta_path中找到。如果输出它的值，可以看到如下的输出：

```
>>> from pprint import


pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
 <class '_frozen_importlib.FrozenImporter'>,
 <class '_frozen_importlib.PathFinder'>]
>>>
```

当执行一条语句比如import fib时，解释器会遍历sys.meta_path中的查询器对象，并调用它们的find_module()方法以此来找到合适的模块加载器。用实验的方式来观察这一过程会对我们的理解有所帮助，因此我们这里定义如下的类并尝试做以下的操作：

```
>>> class Finder


:
...


     def
```

```
find_module(self, fullname, path):
...

            print

('Looking for', fullname, path)
...

            return

None
...


>>> import sys


>>> sys.meta_path.insert(0, Finder()) # Insert as first entry


>>> import math


Looking for math None
>>> import types


Looking for types None
>>> import threading


Looking for threading None
Looking for time None
Looking for traceback None
```

```
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

注意在每个import操作中find_module()方法是如何被触发执行的。在这个方法中，参数path的作用是用来处理包的。当导入的是包时，参数path表示的是包的__path__属性中列出的目录列表。需要检查这些路径来找出包中的子模块。例如，注意xml.etree和xml.etree.ElementTree的路径设定：

```
>>> import xml.etree.ElementTree



Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/
Looking for warnings None
Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
>>>
```

查询器对象在sys.meta_path中的位置是至关重要的。做个试验，将查询器对象从列表头移动到列表尾部，然后尝试做几个导入操作：

```
>>> del


sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request



>>> import datetime
```

现在看不到任何输出了，因为现在的导入操作被sys.meta_path中的其他条目处理了。在这种情况下，只有在导入并不存在的模块时才会触发我们自定义的查询器：

```
>>> import fib



Looking for fib None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast



Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'xml.superfast'
>>>
```

我们可以安装一个查询器以捕获未知的模块，这一事实正是本节中给出的UrlMetaFinder类的核心所在。在sys.meta_path的尾部添加一个UrlMetaFinder实例，把它当做导入操作的最后一道保险。如果请求的模块名无法被其他导入机制所定位，那么就由这个查询器来负责处理。当处理包的导入时还需要注意一些事项。具体来说就是，对于path参数的值，我们需要检查看它是否以我们注册到查询器中的URL开头。如果不是，那么子模块肯定属于其他查询器负责处理的范畴，这里就应该忽略。

对于包的附加处理可以在UrlPackageLoader类中找到。这个类不是去导入包的名称，而是尝试加载底层的*__init__.py*文件，最后它还会设定模块的__path__属性。最后这一步是非常关键的，因为当加载包的子模块时，这个设定的值会传递给接下来的find_module()调用。

基于路径的import钩子是对这些思想的一种扩展，只是其中的机制有所不同。如你所知，sys.path是一个路径列表，其中保存的是Python查询模块的路径。例如：

```
>>> from pprint import

pprint
```

```
>>> import sys



>>> pprint(sys.path)
['',
 '/usr/local/lib/python33.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/usr/local/lib/...3.3/site-packages']
>>>
```

sys.path中的每一个条目都会同一个查询器对象关联起来。我们可以通过打印sys.path_importer_cache来查看这些查询器：

```
>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
 '/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3
 '/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3
 '/usr/local/lib/python3.3/collections': FileFinder('...python3.3
 '/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/e
 '/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3
 '/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3
 '/usr/local/lib/python3.3/site-packages': FileFinder('...python3
 '/usr/local/lib/python33.zip': None}
>>>
```

sys.path_importer_cache比sys.path要大的多，因为前者会针对所有已知代码将被加载的目录记录下相应的查询器。这包括了包中的子目录，而这些信息通常是不包含在sys.path中的。

当执行import fib时，sys.path中的目录会按顺序逐个接受检查。对于每个目录，名称fib会被传递给sys.path_importer_cache中与目录相关联的查询器。我们也可以创建自己的查询器，并将其添加到sys.path_importer_cache中。比如下面这个实验：

```
>>> class Finder


:
...


    def


find_loader(self, name):
...


            print


('Looking for', name)
...


            return


(None, [])
...



>>> import sys



>>> # Add a "debug" entry to the importer cache
```

```
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Add a "debug" directory to sys.path



>>> sys.path.insert(0, 'debug')
>>> import threading



Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize
Looking for token
>>>
```

这里，我们以debug为名称安装了一个新的缓存条目，并把debug作为sys.path的第一个条目。在之后所有的导入动作中，就会发现自己定义的查询器都会被触发执行。但是，由于它只会返回(None, [])，因此处理流程只是简单地继续前往下一个条目执行。

sys.path_importer_cache中的内容是由保存在sys.path_hooks中的一系列函数来控制的。尝试做下面这个实验，它会清空缓存，并添加一个新的路径检查函数到sys.path_hooks中去：

```
>>> sys.path_importer_cache.clear()
```

```
>>>

    def

check_path(path):
...

        print

('Checking', path)
...

        raise ImportError

()
...



>>> sys.path_hooks.insert(0, check_path)
>>> import fib



Checked debug
Checking .
Checking /usr/local/lib/python33.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>
```

可以看到，针对sys.path中的每一个条目，check_path()函数都会得到调用。但是由于会产生ImportError异常，因此实际上什么也没发生（只是跳转到sys.path_hooks中的下一个函数继续执行检查）。

利用sys.path的处理机制，我们可以安装一个自定义的路径检查函数来查找特定的文件名模式，比如URL。示例如下：

```
>>> def


check_url(path):
...


     if


path.startswith('http://'):
...


             return


Finder()
...


     else


:
...
```

```
            raise ImportError

()
...

>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib

Looking for fib             # Finder output!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Notice installation of Finder in sys.path_importer_cache

>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>
```

这就是本节最后这部分内容的核心工作原理。从本质上说，就是在sys.path_hooks中安装一个用来寻找URL的自定义路径检查函数。当遇到这个自定义检查函数时，会产生一个新的UrlPathFinder实例并将其安装到sys.path_importer_cache中。从此之后，对于所有的导入语句，只要在遍历sys.path的过程中遇到这个部分，就会尝试使用自定义的查询器了。

基于路径的导入器在处理包的时候多少需要一些技巧，与find_loader()方法的返回值也有关系。对于简单的模块来说，find_loader()返回一个元组(loader, None)，这里的loader是将要导入这个模块的加载器实例。

对于一个普通的包来说，find_loader()返回一个元组(loader, path)，这里的loader是将要导入这个包的加载器实例（并且会执行__*init*__*.py*），而path是一个目录列表，这些目录会组成包的__path__属性的初始值。比方说，如果URL是http://localhost:15000，并且某个用户执行了import grok 语句，find_loader()返回的路径就会是['http://localhost: 15000/grok']。

find_loader()还必须负责处理命名空间包（见10.5节）的情况。命名空间包是一个合法的包，其目录名存在但其中并不包含__*init*__*.py* 文件。对于这种情况，find_loader()必须返回一个元组(None, path)，这里的path是一个目录列表，这些目录组成了包的__path__属性，并和普通的包一样，认为在这些目录中定义有__*init*__*.py* 文件（实际并不存在）。对于这种情况，import导入机制会继续检查sys.path中的目录。如果找到了更多的命名空间包，那么所有找到的结果路径都会连接在一起以形成一

个最终的命名空间包。有关命名空间包的更多信息请参见10.5节。

在我们的解决方案中，在处理包的时候还运用了递归的思想，虽然看起来并不明显但它同样能够工作。所有的包都包含一个内部的路径设置，这可以在__path__属性中找到。示例如下：

```
>>> import xml.etree.ElementTree



>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>
```

正如我们提到过的，__path__的设置是由find_loader()方法的返回值来控制的。但是，之后对__path__的处理也是由sys.path_hooks中的函数来处理的。因此，当加载包中的子模块时，__path__中的条目是由handle_url()函数来负责检查的。这会导致创建出新的UrlPathFinder实例并添加到sys.path_importer_cache中。

在我们的实现中，最后一个棘手的部分是考虑handle_url()函数的行为以及它同内部使用的函数_get_links()之间的交互。如果我们的查询器实现中

用到了其他的模块（例如urllib.request），那么有可能这些模块会在查询器工作的过程中发起进一步的模块导入请求。这就会导致handle_url()和查询器的其他部分以循环递归的形式执行下去。为了应对这种可能，我们给出的实现中对已经创建出的查询器维护了一个缓存对象（每条URL对应一个缓存）。这就避免了重复创建查询器的问题。此外，下面的代码片段确保了查询器在获取初始链接的过程中不会去响应任何导入请求：

```
# Check link cache

if

self._links is

None:
    self._links = [] # See discussion

    self._links = _get_links(self._baseurl)
```

在其他的实现中可能不需要做上述检查，但是对于这个涉及 URL 的例子，这么做是必需的。

最后，查询器中的invalidate_caches()是一个实

用的方法，当源代码需要发生改变时用来清除内部的缓存。当用户调用importlib.invalidate_caches()时会触发该方法执行。如果想让URL导入器重新读取链接列表，可以使用这个方法。这么做可能是为了能够访问到新添加的文件。

对比以上两种方法（修改sys.meta_path或者使用path钩子）有助于站在更高层的角度来看待问题。利用sys.meta_path安装的导入器可以自由地以任何它们所希望的方式来处理模块。例如，它们可以从数据库中取出模块加载，或者以根本不同于普通的模块/包处理的方式来完成导入。这种自由同样意味着这样的导入器需要做更多的簿记（bookkeeping）和内部管理工作。这也解释了为何在UrlMetaFinder的实现中需要自己实现链接缓存、加载器以及其他一些细节。另一方面，基于路径的钩子方法则同sys.path的处理联系得更为紧密。由于同sys.path的关系紧密，以这种扩展方式加载的模块在特征上倾向于与程序员通常使用的普通模块和包相同。

假设你的大脑现在还没有完全爆炸，理解和试验本节中技术的关键方法可能就是添加日志记录了。我们可以开启日志功能并尝试如下的试验：

```
>>> import logging
```

```
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python33.zip. [No]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
>>>
```

最后但同样重要的是，睡前在枕头下备好PEP 302（http://www.python.org/dev/peps/pep-0302 ）和importlib的文档，花些时间读一读它们也许是很明

智的。

10.12　在模块加载时为其打补丁

10.12.1　问题

我们想对已有的模块打补丁或对其中的函数添加装饰器。但是，我们只希望当模块实际得到导入的时候才这么做，之后再在别处使用。

10.12.2　解决方案

这个问题的关键在于我们想针对正在加载的模块执行响应操作。当某个模块得到加载时，也许我们想触发某种形式的回调函数来通知这一事实。

这个问题可以使用10.11节中讨论过的import钩子机制来解决。下面是一种可能的解决方案：

```
# postimport.py

import importlib

import sys
```

```
from collections import

defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder

:
    def

__init__(self):
        self._skip = set()

    def

find_module(self, fullname, path=None):
        if

fullname in

self._skip:
            return

None
        self._skip.add(fullname)
        return

PostImportLoader(self)

class PostImportLoader

:
    def
```

```
__init__(self, finder):
        self._finder = finder

    def


load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for


func in


_post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return


module

def


when_imported(fullname):
    def


decorate(func):
        if


fullname in


sys.modules:
            func(sys.modules[fullname])
        else


:
            _post_import_hooks[fullname].append(func)
```

```
        return


func
    return


decorate

sys.meta_path.insert(0, PostImportFinder())
```

要使用这份代码，就要用到when_imported()装饰器。示例如下：

```
>>> from postimport import


when_imported
>>> @when_imported('threading')
... def


warn_threads(mod):
...     print


('Threads? Are you crazy?')
...
>>>
>>> import threading



Threads? Are you crazy?
>>>
```

作为一个更加实际的例子，也许想在已有的定义上添加装饰器，比如：

```
from functools import

wraps
from postimport import

when_imported

def

logged(func):
    @wraps(func)
    def

wrapper(*args, **kwargs):
        print

('Calling', func.__name__, args, kwargs)
        return

func(*args, **kwargs)
    return

wrapper

# Example

@when_imported('math')
def
```

```
add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)
```

10.12.3　讨论

本节依赖于10.11节中讨论过的import钩子技术，做了少许修改。

首先，@when_imported装饰器的作用是注册在导入时需要触发执行的处理函数。这个装饰器检查sys.modules，看某个模块是否已经加载了。如果是，则立刻调用处理函数。否则就将处理函数添加到_post_import_hooks字典中去。定义_post_import_hooks的目的只是简单地用来收集所有的已经针对每个模块所注册的处理对象。原则上，对于一个给定的模块可以注册多个处理程序。

在导入模块之后，要触发_post_import_hooks中挂起的操作，PostImportFinder类的实例就要安装到sys.meta_path 中的首元素位置上。如果回顾一下10.11 节就会知道，sys.meta_path中包含一列用来定位模块位置的查询器对象。通过将PostImportFinder类的实例安装到列表的首元素位置，那么它就能捕获所有的模块导入动作。

但是在本节中，PostImportFinder的作用不是用来加载模块，而是触发相应的处理流程来完成整个导入动作。要做到这一点，实际的导入被委托给sys.meta_path中的其他查询器来完成。不要尝试直接实现这一步骤，相反，我们是在PostImportLoader类中递归调用函数imp.import_module()来完成的。为了避免出现无限递归的情况，我们在PostImportFinder类中维护了一个集合，其中包含所有当前正处于加载过程中的模块。如果某个模块名属于这个集合，PostImportFinder只会简单地忽略它。这就是导致import请求会传递给sys.meta_path中其他查询器处理的原因。

在一个模块已经通过imp.import_module()加载之后，所有当前注册到_post_import_hooks中的处理例程都会以新加载的模块作为参数得到调用。从这一刻开始，处理例程就可以自由地对模块做任何想做的操作了。

本节所展示的方法中一个主要的特性就是对模块的打补丁操作是以无缝的方式进行的，与所感兴趣的模块的实际位置和加载方式无关。只需要简单地编写一个处理函数并用@when_imported()进行装饰，从那一刻起所有的操作就能魔法般地工作起来。

本节中需要注意的一个问题是对于已经使用imp.reload()显式重新加载的模块，本节给出的方法是无效的。也就是说，如果重新加载一个之前加载过的模块，处理例程是不会再次得到触发的（我们有了更多的理由不要在生产环境中使用reload()）。而另一方面，如果从sys.modules中删除模块然后再重新导入，就会发现处理例程会再次触发执行。

更多有关post-import钩子的信息可以在PEP 369（http://www.python.org/dev/peps/pep-0369 ）中找到。在写作本节时，这份PEP已经被作者撤销了，原因是它同当前的importlib模块的实现相比显得过时了，但是利用本节所展示的方法实现自己的解决方案已经足够简单了。

10.13　安装只为自己所用的包

10.13.1　问题

我们想安装一个第三方的包，但是没有权限在系统Python中安装其他的包。另一种情况是，也许我们只想安装一个给自己使用的包，而不是给系统中的所有用户使用。

10.13.2　解决方案

Python有一个用户级的安装目录，通常位于类似*~./local/lib/python3.3/site-packages* 这样的目录下。要让包强制安装到这个目录下，只要在安装命令后添加--user选项即可。示例如下：

```
python3 setup.py install –user
```

或者

```
pip install --user packagename
```

用户级的*site-package* 目录通常会在sys.path中出现，且位于系统级的*site-package* 目录之前。因此，采用这种技术安装的包比已经安装到系统中的包优先级要高（尽管并不会总是这样，这取决于第三方包管理工具的具体行为，比如distribute或pip）。

10.13.3　讨论

一般来说，包会被安装到系统级的 *site-package* 目录下，可以在例如*/usr/local/lib/python3.3/site-packages* 的位置上找到。但是安装到系统级的目录下通常都需要有管理员权限，并且要使用sudo命令。就算我们有权限执行这样的命令，使用sudo安装一个新的且没有经过实践证明的包也可能会给我们带来麻烦。

把包安装到用户级的目录中通常是一种有效的规避方案，这么做允许我们创建一个自定义的安装。

另一种解决方案是可以创建一个虚拟环境，这正是我们在下一节要讨论的主题。

10.14　创建新的Python环境

10.14.1　问题

我们想创建一个新的Python环境，在新环境中可以自由地安装模块和包。但是，我们并不想为此安装一个新的Python拷贝或者做出任何可能会影响到系统级Python安装的修改。

10.14.2　解决方案

可以通过pyvenv命令创建一个新的“虚拟”环境。这个命令被安装到同Python解释器一样的目录中，在Windows下可能位于*Scripts* 目录下。这里有一个示例：

```
bash % pyvenv Spam
bash %
```

提供给pyvenv的名称就是将要创建的目录名称。创建好之后，*Spam* 目录看起来是这样的：

```
bash % cd Spam
bash % ls
bin         include          lib           pyvenv.cfg
```

```
bash %
```

在*bin* 目录下，我们会发现有一个可使用的Python解释器。示例如下：

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct 6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more informa
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
 '/usr/local/lib/python33.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/Users/beazley/Spam/lib/python3.3/site-packages']
>>>
```

这个解释器的核心特征就是它的*site-package* 目录已经被设定为同新创建的环境相关联了。如果决定安装第三方的包，那么它们就会被安装到这里，而不是普通的系统级*site-package* 目录中。

10.14.3　讨论

创建虚拟环境大部分的原因都是为了安装和管理第三方的包。可以在示例中看到，sys.path变量中包含的目录来自于普通的系统级Python，但是*site-*

package 目录已经被重新定位到新的目录上了。

有了新的虚拟环境，下一步通常是安装一个包管理器，比如distribute或者pip。当安装这类工具和其他的包时，只需确保使用的是虚拟环境中的解释器即可。这样的话，安装的包应该就会放在新创建的*site-packages* 目录下了。

尽管虚拟环境看起来好像是Python安装的一份拷贝，但它实际上只是由几个文件和一些符号链接所组成。所有的标准库文件和解释器执行文件都来自于原来的Python安装包。因此，创建这样的环境非常简单方便，几乎不占用什么系统资源。

默认情况下，虚拟环境是完全干净且不包含任何第三方插件的。如果想将已经安装过的包引入，使其作为虚拟环境的一部分，那么可以使用选项--system-site-packages来创建虚拟环境。示例如下：

```
bash % pyvenv --system-site-packages Spam
bash %
```

有关 pyvenv 和虚拟环境的更多信息可以在PEP 405（http://www.python.org/dev/peps/ pep-0405）中找到。

10.15 发布自定义的包

10.15.1 问题

我们编写了一个有用的库，想将其分发给其他人使用。

10.15.2 解决方案

如果打算将代码发布出去，首先要做的就是为自己的库起一个独一无二的名称并清理代码的目录结构。例如，一个典型的程序库结构看起来大致是这样的：

```
projectname/
    README.txt
    Doc/
        documentation.txt
    projectname/
        __init__.py
        foo.py
        bar.py
        utils/
            __init__.py
            spam.py
            grok.py
    examples/
        helloworld.py
        ...
```

要使得包能够发布出去，首先编写一个*setup.py* 文件，看起来是这样的：

```
# setup.py
from distutils.core import setup

setup(name='projectname',
      version='1.0',
      author='Your Name',
      author_email='you@youraddress.com',
      url='http://www.you.com/projectname',
      packages=['projectname', 'projectname.utils'],
)
```

接下来要创建一个*MANIFEST.in* 文件，并在其中列出各种希望包含在包中的非源代码文件：

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

确保*setup.py* 和*MANIFEST.in* 文件位于包的顶层目录。一旦做完这些，应该就能够通过命令来创建一个源代码级的分发包了：

```
% bash python3 setup.py sdist
```

根据不同的系统平台，这么做会创建出像*projectname-1.0.zip* 或者*projectname-1.0.tar.gz* 这样的文件。如果一切顺利，这个文件就可用来发布给其他人或者上传到Python Package Index（http://pypi.python.org ）上了。

10.15.3　讨论

对于纯Python代码来说，编写一个简单的*setup.py* 文件通常是很直接的。但其中一个潜在的问题是我们必须手动列出包中的每一个子目录。一个常见的错误就是只列出了包的顶层目录而忘记包含进包中的子模块。这就是为什么在*setup.py* 中对包的规格说明里包含了列表packages=['projectname', 'projectname.utils']的原因。

大多数Python程序员都知道，如今有许多第三方的包管理工具，包括安装、发布相关的，等等。这些第三方工具中有一些可用来替代标准库中的distutils库。请注意，如果要依赖这些包，那么用户可能没法安装使用我们的软件，除非他们也安装了所需的包管理工具。基于此，只要我们尽可能让事情变得简单，那就几乎不会出什么错误。最低要求是请确保代码可以通过使用标准的Python 3安装方式来安装。如果还有别的安装包可用，那么可以将它们作为可选项以支持额外的功能。

打包和发布涉及C语言扩展的代码则会变得复杂得多。在第15章有关C语言扩展的章节中谈到了一些关于此的细节。具体请参见15.2节。

第11章　网络和Web编程

本章涵盖了在网络应用和分布式应用中使用Python的各种主题。主题可划分为使用Python编写客户端程序来访问已有的服务，以及使用Python实现网络服务端程序。本章也提到了编写代码使多个解释器协同工作或者互相通信的常见技术。

11.1　以客户端的形式同HTTP服务交互

11.1.1　问题

我们需要以客户端的形式通过HTTP协议访问多种服务。比如，下载数据或者同一个基于REST的API进行交互。

11.1.2　解决方案

对于简单的任务来说，使用urllib.request模块通常就足够了。比方说，要发送一个简单的HTTP GET请求到远端服务器上，可以这样做：

```
from urllib import

request, parse
# Base URL being accessed

url = 'http://httpbin.org/get'
# Dictionary of query parameters (if any)
```

```
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Encode the query string



querystring = parse.urlencode(parms)
# Make a GET request and read the response



u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

如果需要使用POST方法在请求主体（request body）中发送查询参数，可以将参数编码后作为可选参数提供给urlopen()函数，就像这样：

```
from urllib import


request, parse

# Base URL being accessed



url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
```

```
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Encode the query string



querystring = parse.urlencode(parms)

# Make a POST request and read the response



u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

如果需要在发出的请求中提供一些自定义的HTTP头，比如修改user-agent字段，那么可以创建一个包含字段值的字典，并创建一个Request实例然后将其传给urlopen()。示例如下：

```
from urllib import


request, parse
...

# Extra headers



headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
```

```
}

req = request.Request(url, querystring.encode('ascii'), headers=h

# Make a request and read the response



u = request.urlopen(req)
resp = u.read()
```

如果需要交互的服务比上面的例子都要复杂，也许应该去看看requests库（http://pypi.python.org/pypi/requests）。比如，下面这个示例采用requests库重新实现了上面的操作：

```
import requests



# Base URL being accessed



url = 'http://httpbin.org/post'
# Dictionary of query parameters (if any)



parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}
```

```
# Extra headers



headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Decoded text returned by the request



text = resp.text
```

关于requests库，一个值得一提的特性就是它能以多种方式从请求中返回响应结果的内容。从上面的代码来看，resp.text带给我们的是以Unicode解码的响应文本。但是，如果去访问resp.content，就会得到原始的二进制数据。另一方面，如果访问resp.json，那么就会得到JSON格式的响应内容。

下面这个示例利用requests库来发起一个HEAD请求，并从响应中提取出一些HTTP头数据的字段：

```
import requests


```

```
resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

下面的示例使用requests库通过基本的认证在Python Package Index（也就是pypi）上执行了一个登录操作：

```
import requests



resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user','password'))
```

下面的示例使用requests库将第一个请求中得到的HTTP cookies传递给下一个请求：

```
import requests



# First request
```

```
resp1 = requests.get(url)
...
# Second requests with cookies received on first requests



resp2 = requests.get(url, cookies=resp1.cookies)
```

最后但也同样重要的是，下面的例子使用requests库来实现内容的上传：

```
import requests



url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

11.1.3　讨论

对于确实很简单的HTTP客户端代码，通常使用内建的urllib模块就足够了。但是，如果要做的不仅仅只是简单的GET或POST请求，那就真的不能再依赖它的功能了。这时候就是第三方模块比如requests大显身手的时候了。

举个例子，如果我们决定坚持使用标准的程序库而不考虑像requests这样的第三方库，那么也许就不得不使用底层的http.client模块来实现自己的代码。比方说，下面的代码展示了如何执行一个HEAD请求：

```
from http.client import


HTTPConnection
from urllib import


parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print


('Status', resp.status)
for


name, value in


resp.getheaders():
    print


(name, value)
```

　　同样地，如果必须编写涉及代理、认证、cookies以及其他一些细节方面的代码，那么使用urllib就显得特别别扭和啰嗦。比方说，下面这个示例实现在Python package index上的认证：

```
import urllib.request



auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi','http://pypi.python.org','username','pa
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?:action=
u = opener.open(r)
resp = u.read()

# From here. You can access more pages using opener



...
```

　　坦白说，所有这些操作在requests库中都变得简单得多。

　　在开发过程中测试HTTP客户端代码常常是很令人沮丧的，因为所有棘手的细节问题都需要考虑（例如cookies、认证、HTTP头、编码方式等）。要完成这些任务，考虑使用httpbin服务

（http://httpbin.org ）。这个站点会接收发出的请求，然后以JSON的形式将响应信息回传回来。下面是一个交互式的例子：

```
>>> import requests



>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

在要同一个真正的站点进行交互前，先在httpbin.org这样的网站上做实验常常是可取的办法。尤其是当我们面对3次登录失败就会关闭账户这样的风险时尤为有用（不要尝试自己编写HTTP认证客户端来登录你的银行账户）。

尽管本节没有涉及，requests库还对许多高级的HTTP客户端协议提供了支持，比如OAuth。requests模块的文档（http://docs.python-requests.org ）质量很高（坦白说比在这短短一节的篇幅中所提供的任何信息都好），可以参考文档以获得更多的

信息。

11.2　创建一个TCP服务器

11.2.1　问题

我们想实现一个通过TCP协议同客户端进行通信的服务器。

11.2.2　解决方案

创建TCP服务器的一种简单方式就是利用socketserver库。比如，下面是一个简单的echo服务示例：

```
from socketserver import


BaseRequestHandler, TCPServer

class EchoHandler


(BaseRequestHandler):
    def


handle(self):
        print


('Got connection from', self.client_address)
        while
```

```
True:
            msg = self.request.recv(8192)
            if not

msg:
                break

            self.request.send(msg)

if

__name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在这份代码中，我们定义了一个特殊的处理类，它实现了一个handle()方法来服务于客户端的连接。这里的request属性就代表着底层的客户端socket，而client_address中包含了客户端的地址。

要测试这个服务端程序，首先运行这个脚本，然后打开另一个Python进程并将其连接到服务端上：

```
>>> from socket import

socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
```

```
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>
```

在许多情况下，定义一个类型稍有不同的处理类可能会更加简单。下面的示例使用StreamRequestHandler作为基类，给底层的socket加上了文件类型的接口：

```
from socketserver import


StreamRequestHandler, TCPServer

class EchoHandler


(StreamRequestHandler):
    def


handle(self):
        print


('Got connection from', self.client_address)
        # self.rfile is a file-like object for reading



        for
```

```
line in

self.rfile:
            # self.wfile is a file-like object for writing

            self.wfile.write(line)

if

__name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

11.2.3　讨论

socketserver模块使得创建简单的TCP服务器相对来说变得容易了许多。但是，应该要注意的是，默认情况下这个服务器是单线程的，一次只能处理一个客户端。如果想处理多个客户端，可以实例化ForkingTCPServer或者ThreadingTCPServer对象。示例如下：

```
from socketserver import

ThreadingTCPServer
...

if
```

```
__name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

多进程和多线程服务器的问题在于它们会针对每一个客户端连接创建一个新的进程或线程。但是允许连接的客户端数量是没有上限的，因此一个怀有恶意的黑客可能会同时发起海量的连接使你的服务器挂掉。

如果需要考虑这个因素，则可以创建一个预先分配好的工作者线程或进程池。要做到这一点，我们首先创建一个普通的（非多线程/多进程）服务器实例，但是之后在多个线程中调用serve_forever()方法。示例如下：

```
...
if


__name__ == '__main__':
    from threading import


Thread
    NWORKERS = 16
    serv = TCPServer(('', 20000), EchoHandler)
    for


n in
```

```
range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

一般来说，TCPServer在实例化时就会绑定并激活底层的socket。但是，有时候我们可能会想通过设定socket选项来调整底层socket的行为。要做到这一点，可以提供bind_and_activate=False参数，就像这样：

```
if

__name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=
    # Set up various socket options

    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR
    # Bind and activate

    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

上面给出的socket选项实际上是一种非常常见的设置。它允许服务器重新对之前使用过的端口号进行绑定。由于这个设置实在是太常用了，因此在TCPServer类中也提供了一个完成相同功能的类变量，在实例化服务器之前设定它即可，如下所示：

```
...
if


__name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在这个解决方案中，我们展示了两个不同的基类（BaseRequestHandler和StreamRequesthandler）。StreamRequestHandler类实际上要更灵活一些，而且可以通过指定额外的类变量来提供一些功能。比如：

```
import socket



class EchoHandler


(StreamRequestHandler):
    # Optional settings (defaults shown)
```

```
    timeout = 5                                 # Timeout on all socket o
    rbufsize = -1                              # Read buffer size
    wbufsize = 0                              # Write buffer size
    disable_nagle_algorithm = False          # Sets TCP_NODELAY so
    def
handle(self):
        print
('Got connection from', self.client_address)
        try
:
            for
line in
self.rfile:
                # self.wfile is a file-like object for writing
                self.wfile.write(line)
        except
```

```
socket.timeout:
            print

('Timed out!')
```

最后，应该要指出的是大部分Python中的高级网络模块（例如HTTP、XML-RPC等）都是在socketserver的功能之上构建的。也就是说，直接使用socket库来实现服务器也并不会太困难。下面这个简单的示例就是直接通过socket来编写服务器程序:

```
from socket import

socket, AF_INET, SOCK_STREAM

def

echo_handler(address, client_sock):
    print

('Got connection from {}'.format(address))
    while

True:
        msg = client_sock.recv(8192)
        if not
```

```
msg:
            break



        client_sock.sendall(msg)
    client_sock.close()

def



echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while



True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if



__name__ == '__main__':
    echo_server(('', 20000))
```

11.3 创建一个UDP服务器

11.3.1 问题

我们想实现一个采用UDP协议同客户端进行通信的服务器。

11.3.2 解决方案

同TCP一样，利用socketserver库也能很容易地创建出UDP服务器。比如，下面有一个简单的时间服务器程序：

```
from socketserver import


BaseRequestHandler, UDPServer
import time




class TimeHandler


(BaseRequestHandler):
    def


handle(self):
        print
```

```
('Got connection from', self.client_address)
        # Get message and client socket



        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if


__name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
serv.serve_forever()
```

同上一节类似，这里需要定义一个特殊的处理类，其中要实现一个handle()方法来处理客户端的连接。这里的request属性是一个元组，包含了这个服务器收到的数据报以及代表底层的socket对象。client_address包含的是客户端的地址。

要测试这个服务器程序，先运行上面的脚本，然后另外打开一个Python进程并向服务端程序发送消息：

```
>>> from socket import


socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
```

```
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

11.3.3 讨论

一个典型的UDP服务器程序会接收到发自客户端的数据报（消息）以及客户端的地址。如果服务端要响应请求，它就发回一个数据报给客户端。对于数据报的发送和传输，应该使用socket对象的sendto()和recvfrom()方法。尽管传统的send()和recv()方法也能工作，但是在UDP通信中前面两种方法更为常用一些。

由于UDP通信底层不需要建立连接，因此UDP服务器常常比TCP服务器要容易编写得多。但是同时UDP也是不可靠的（例如，由于没有建立连接，消息可能会丢失）。因此，如何处理消息丢失的任务就交给了你自己。这个主题超出了本书的范围，但是如果可靠性对于你的程序来说很重要，一般来说就要引入序列号、重传、超时以及其他的机制来确保传输的可靠性。UDP常用在对可靠性传输要求不那么高的应用中。例如，在类似多媒体流应用以及游戏中常会用到UDP，因为在这类应用中根本不会倒退回去试图重传某个丢失的数据包（程序会简

单地忽略丢弃的包，然后继续运行）。

UDPServer类也是单线程的，这意味着它一次只能处理一个请求。在实践中，这个问题与TCP连接相比要小很多。但是如果要实现并发操作，可以实例化ForkingUDPServer或者ThreadingUDPServer：

```
from socketserver import

ThreadingUDPServer
...
if

__name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

直接通过socket来实现UDP服务器同样也不复杂。示例如下：

```
from socket import

socket, AF_INET, SOCK_DGRAM
import time
```

```
def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```

11.4　从CIDR地址中生成IP地址的范围

11.4.1　问题

我们有一个类似于“123.45.67.89/27”这样的CIDR（Classless InterDomain Routing）网络地址，我们想生成由该地址可表示的全部IP地址的范围（例如，“123.45.67.64”,“123.45.67.65”...，“123.45.67.95”

11.4.2　解决方案

利用ipaddress模块可轻松处理这样的计算。示例如下：

```
>>> import ipaddress



>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for


a in
```

```
net:
...

    print

(a)
...



123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...



123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/1
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for

a in

net6:
...

    print

(a)
...
```

```
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

network对象同样也支持像数组那样的索引操作。示例如下：

```
>>> net.num_addresses
32
>>> net[0]
IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>
```

此外，还可以执行检查成员归属的操作：

```
>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in


net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in
```

```
net
False
>>>
```

IP地址加上网络号可以用来指定一个IP接口（interface）。比如：

```
>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>
```

11.4.3　讨论

ipaddress模块中有一些类可用来表示IP地址、网络对象以及接口。如果要编写代码以某种方式来操作网络地址的话（比如解析、打印、验证等），这就显得非常有帮助了。

需要注意的是，ipaddress模块同其他网络相关的模块比如socket库之间的交互是有局限性的。特别是，通常不能用IPv4Address的实例作为地址字符串的替代。相反，必须显式地通过str()将其转换为

字符串。示例如下：

```
>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import


socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>
```

请参阅“ipaddress模块介绍”（http://docs.python.org/3/howto/ipaddress.html ）一文，以获得更多的信息和高级用法的示例。

11.5　创建基于REST风格的简单接口

11.5.1　问题

我们希望通过一个基于REST风格的简单接口来对程序实现远程控制或交互。但是，我们又不想为此去安装一个成熟的Web编程框架。

11.5.2　解决方案

构建基于REST风格的接口最简单的方式之一就是根据WSGI规范（在PEP 3333中描述，地址为http://www.python.org/dev/peps/pep-3333 ）创建一个小型的库。示例如下：

```
# resty.py

import cgi

def
```

```
notfound_404(environ, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/pla
    return


[b'Not Found']

class PathDispatcher


:
    def


__init__(self):
        self.pathmap = { }

    def


__call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for


key in


params }
        handler = self.pathmap.get((method,path), notfound_404)
        return


handler(environ, start_response)

    def


register(self, method, path, function):
        self.pathmap[method.lower(), path] = function
        return
```

```
function
```

要使用这个调度器，只用编写不同的处理函数即可，比如：

```
import time



_hello_resp = '''\



<html>
  <head>
    <title>Hello {name}</title>
  </head>
  <body>
    <h1>Hello {name}!</h1>
  </body>
</html>'''

def


hello_world(environ, start_response):
    start_response('200 OK', [ ('Content-type','text/html')])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield


resp.encode('utf-8')

_localtime_resp = '''\
```

```
<?xml version="1.0"?>
<time>
    <year>{t.tm_year}</year>
    <month>{t.tm_mon}</month>
    <day>{t.tm_mday}</day>
    <hour>{t.tm_hour}</hour>
    <minute>{t.tm_min}</minute>
    <second>{t.tm_sec}</second>
</time>'''

def


localtime(environ, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml
    resp = _localtime_resp.format(t=time.localtime())
    yield


resp.encode('utf-8')

if


__name__ == '__main__':
    from resty import


PathDispatcher
    from wsgiref.simple_server import


make_server

    # Create the dispatcher and register functions



    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)
```

```
    # Launch a basic server



    httpd = make_server('', 8080, dispatcher)
    print


('Serving on port 8080...')
    httpd.serve_forever()
```

要测试这个服务程序，可以通过浏览器或者使用urllib来完成交互。示例如下：

```
>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print


(u.read().decode('utf-8'))
<html>
  <head>
    <title>Hello Guido</title>
  </head>
  <body>
    <h1>Hello Guido!</h1>
  </body>
</html>
>>> u = urlopen('http://localhost:8080/localtime')
>>> print


(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>
  <year>2012</year>
  <month>11</month>
  <day>24</day>
  <hour>14</hour>
  <minute>49</minute>
```

```
  <second>17</second>
</time>
>>>
```

11.5.3　讨论

在基于REST风格的接口中，一般来说就是在编写响应常见HTTP请求的程序。但是，与一个成熟的网站不同，通常我们只是在来回推送数据。这个数据可能会以各种标准的格式进行编码，比如XML、JSON或者CSV。尽管看起来似乎微不足道，但是以这种方式提供API对于各种各样的应用程序都是非常有用的。

比如，长时间运行的程序可能会用REST风格的API来实现监控或诊断功能。大数据应用可以使用REST风格的接口来构建一个查询/提取数据的系统。REST甚至可以用来控制硬件设备，比如机器人、传感器或者是灯泡。此外，REST API在各式各样的客户端编程环境中都得到了很好的支持，比如JavaScript、Android、iOS等。因此，有了这样的接口能够鼓励人们开发出更加复杂的应用来使用你的接口代码。

要实现一个简单的REST风格的接口，通常只

要根据Python的WSGI标准来做就可以了。标准库是支持WSGI的，而且大多数第三方的Web框架也支持。因此，如果采用WSGI标准的话，我们的代码使用起来将变得非常灵活。

在WSGI中，应用程序是以一个接受如下调用形式的可调用对象来实现的：

```
import cgi



def

wsgi_app(environ, start_response):
    ...
```

参数environ是一个字典，其中需要包含的值参考了许多Web服务器比如Apache所提供的CGI接口的启发。要提取出不同的字段，可以编写这样的代码来实现：

```
def

wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
```

```
    # Parse the query parameters


    params = cgi.FieldStorage(environ['wsgi.input'], environ=envi
    ...
```

示例中展示了一些常见的值。environ['REQUEST_METHOD']表示请求的类型（例如GET、POST、HEAD 等）。environ['PATH_INFO']表示所请求资源的路径。调用cgi.FieldStorage()可以从请求中提取出所提供的查询参数，并将它们放入到一个类似于字典的对象中以供稍后使用。

参数start_response是一个函数，必须调用它才能发起响应。start_response的第一个参数是HTTP结果状态。第二个参数是一个元组序列，以(name, value)这样的形式组成响应的HTTP头。示例如下：

```
def


wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
```

要返回数据，满足WSGI规范的应用程序必须返回字节串序列。这可以通过使用一个列表来完成：

```
def


wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n


')
    resp.append(b'Goodbye!\n


')
    return


resp
```

此外，也可以使用yield作为替代方案：

```
def


wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield
```

```
b'Hello World\n

'
    yield

b'Goodbye!\n

'
```

需要重点强调的是，返回的结果必须使用字节串的形式。如果响应是由纯文本组成的，那就需要先将其编码为字节形式。当然了，这里并没有要求返回的结果是文本，因此可以轻松编写一个应用程序来生成图像。

尽管遵循WSGI规范的应用程序通常都被定义为函数，就如我们的示例那样，但是类实例同样也是可行的，只要它实现了合适的__call__()方法即可。示例如下：

```
class WSGIApplication

:
    def

__init__(self):
        ...
```

```
    def

__call__(self, environ, start_response)
        ...
```

在本节中，我们已经采用这种技术创建了PathDispatcher类。这个调度器只负责管理一个字典，用来将方法和路径的映射关系传递给处理函数，除此之外什么都不做。当有请求到来时，提取出方法和路径然后将其分发给一个处理函数。此外，任何查询变量都会得到解析并放入到字典中以environ['params']的形式保存（这个步骤非常常见，为了避免产生大量重复的代码，在调度器中统一处理是很有意义的）。

要使用这个调度器，只要创建一个实例并注册各种基于WSGI风格的应用程序函数到调度器中即可，就像本节示例中的那样。编写这些函数应该是非常直接的，只要遵循start_response()函数的规则并且以字节串作为输出即可。

当编写这样的函数时，对于字符串模板的使用需要特别小心。没人喜欢和一堆由print()函数、XML以及各式各样的格式化操作杂揉在一起的代码打交道。在我们的解决方案中，采用的方式是定义

三引号式的字符串模板然后在内部使用。这种方式使得之后修改输出的格式变得更加简单了（只要修改模板即可，而不用到处去修改代码）。

最后，使用WSGI的一个重要因素就是实现中没有任何部分是特定于某个具体的Web服务器的。这正是WSGI规范所代表的全部意义——由于WSGI是与服务器以及框架无关的，我们应该可以将自己的应用程序部署到各式各样的服务器之上。在本节中，可以使用下面的代码来测试：

```
if

__name__ == '__main__':
    from wsgiref.simple_server import

make_server

    # Create the dispatcher and register functions

    dispatcher = PathDispatcher()
    ...

    # Launch a basic server

    httpd = make_server('', 8080, dispatcher)
    print

('Serving on port 8080...')
```

```
    httpd.serve_forever()
```

这样会创建出一个简单的服务器，可以用它来检查我们的实现是否都能正常工作。之后，当准备好将应用的规模扩展到更高的层次时，我们就可以修改实现代码让它工作在某个特定的服务器上。

WSGI被有意设计为功能最简化的规范。比如，它不提供任何类似认证、cookies、重定向等高级功能的支持。这些功能由你自己实现并不算困难。但是，如果想得到更多的支持，可以考虑一些第三方的库，比如WebOb（http://webob.org ）或者Paste（http://pythonpaste.org ）。

11.6 利用XML-RPC实现简单的远端过程调用

11.6.1 问题

我们希望能有一种简单的方法可以在远端机器上运行的 Python 程序中执行函数或者方法。

11.6.2 解决方案

也许实现一个远端过程调用机制最简单的方式就是使用XML-RPC了。下面这个例子给出了一个简单的服务器，其中实现了键—值对的存储：

```
from xmlrpc.server import


SimpleXMLRPCServer

class KeyValueServer



:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def


__init__(self, address):
        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True
```

```
        for

name in

self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def

get(self, name):
        return

self._data[name]

    def

set(self, name, value):
        self._data[name] = value

    def

delete(self, name):
        del

self._data[name]

    def

exists(self, name):
        return

name in

self._data
```

```
    def

keys(self):
        return

list(self._data)

    def

serve_forever(self):
        self._serv.serve_forever()

# Example

if

__name__ == '__main__':
    kvserv = KeyValueServer(('', 15000))
    kvserv.serve_forever()
```

下面的代码展示了如何从客户端远程访问服务器：

```
>>> from xmlrpc.client import

ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
```

```
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

11.6.3 讨论

要配置一个简单的远端过程调用服务，可以用XML-RPC来轻松实现。所有要做的就是创建一个服务器实例，通过register_function()方法注册处理函数，然后通过serve_forever()方法加载即可。本节给出的示例把所有的代码集合到了一起，将这些步骤打包到了一个类中，但是实际上并没有这个硬性要求。比如，我们可以创建一个这样的服务器：

```
from xmlrpc.server import


SimpleXMLRPCServer
def


add(x,y):
    return


x+y

serv = SimpleXMLRPCServer(('', 15000))
```

```
serv.register_function(add)
serv.serve_forever()
```

通过XML-RPC暴露出的函数只能处理几种特定类型的数据，比如字符串、数字、列表和字典。至于其他类型的数据则需要做进一步的研究。比方说，如果通过 XML-RPC传递一个实例，则只有它的实例字典会被处理：

```
>>> class Point


:
...


    def


__init__(self, x, y):
...


            self.x = x
...


            self.y = y
...



>>> p = Point(2, 3)
>>> s.set('foo', p)
>>> s.get('foo')
{'x': 2, 'y': 3}
>>>
```

同样地，对二进制数据的处理也和我们期望的方式有所不同：

```
>>> s.set('foo', b'Hello World')
>>> s.get('foo')
<xmlrpc.client.Binary object at 0x10131d410>
>>> _.data
b'Hello World'
>>>
```

作为一般的规则，不应该将XML-RPC服务作为公有API暴露给外部世界。通常，最佳应用场景是在内部网络中，我们可以编写涉及几台不同机器的简单的分布式应用程序。

XML-RPC的缺点在于它的性能。SimpleXMLRPCServer是以单线程来实现的，尽管可以通过11.2节所示的方法配置为以多线程方式运行，但还是不适合用来扩展大型的应用。此外，由于XML-RPC会将所有的数据序列化为XML格式，因此就会比其他的方法要慢一些。但是，这种编码的优势在于许多其他的编程语言都能够理解。使用XML-RPC的话，客户端程序就可以采用Python之外的语言来编写，同样可以访问你的服务。

抛开XML-RPC的局限性不说，如果需要以快速但并不完善（quick and dirty）的方式实现一个远端过程调用系统，那么了解一下XML-RPC还是很值得的。很多时候简单的方案就已经足够好了。

11.7　在不同的解释器间进行通信

11.7.1　问题

我们正运行着多个Python解释器的实例，有可能还是在不同的机器上。我们想通过消息在不同的解释器之间交换数据。

11.7.2　解决方案

如果使用multiprocessing.connection模块，那么在不同的解释器之间实现通信就很简单了。下面是一个实现了echo服务的简单示例：

```
from multiprocessing.connection import


Listener
import traceback




def


echo_client(conn):
    try


:
```

```
        while


True:
            msg = conn.recv()
            conn.send(msg)
    except EOFError


:
        print


('Connection closed')

def


echo_server(address, authkey):
    serv = Listener(address, authkey=authkey)
    while


True:
        try


:
            client = serv.accept()
            echo_client(client)
        except Exception


:
            traceback.print_exc()

echo_server(('', 25000), authkey=b'peekaboo')
```

下面的客户端连接到服务器上并发送各种消息：

```
>>> from multiprocessing.connection import


Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
>>> c.recv()
'hello'
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>
```

和低级的socket不同，这里所有的消息都是完整无损的（每个由send()发送的对象都会通过recv()完整地接收到）。此外，对象都是通过pickle来进行序列化的。因此，任何同pickle兼容的对象都可以在连接之间传递和接收。

11.7.3 讨论

有许多软件包和库都实现了各种形式的消息传递，比如ZeroMQ、Celery等。作为备用方案，我们可能也会倾向于在底层的socket之上实现一个消息层。但是，有时候我们只想要一个简单的解决方案。multiprocessing.connection库正是我们所需要的——只需要使用几个简单的原语（primitive），就

能轻易地将各个解释器联系在一起并且在它们之间交换消息。

如果知道这些解释器会运行在同一台机器上，那么可以利用网络作为替代方案，比如UNIX域socket或者Windows上的命名管道。要通过UNIX域socket创建连接，只要简单地将地址改为文件名即可，示例如下：

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

要通过Windows命名管道创建连接，可以使用下面这样的文件名：

```
s = Listener(r'\\

.\pipe\myconn', authkey=b'peekaboo')
```

作为一般的规则，不应该使用 multiprocessing 模块来实现面向公众型的服务。传递给Client()和Listener()的参数 authkey 在这里是为了帮助认证连接的对端节点。用错误的密钥来建立连接会产生异常。此外，这个模块最好适用于能长时间运行的连接（而不是大量的短连接）。例如，两个解释器可

能会在开始的时候建立一条连接，然后在整个过程中都保持这个连接的活跃。

如果需要对连接实现更多的底层控制，那么就不要使用multiprocessing模块。比方说，如果要支持超时、非阻塞I/O或者任何类似的特性，那么最好使用另一个不同的库或者直接在socket之上实现这些特性。

11.8　实现远端过程调用

11.8.1　问题

我们想在socket、multiprocessing.connection或者ZeroMQ这样的消息传递层之上实现简单的远端过程调用（RPC）。

11.8.2　解决方案

通过将函数请求、参数以及返回值用pickle进行编码，然后在解释器之间传递编码过的pickle字节串，RPC是很容易实现的。下面的示例是一个简单的RPC处理例程，可以将其纳入到一个服务器程序中使用。

```
# rpcserver.py

import pickle

class RPCHandler

:
```

```
    def


__init__(self):
        self._functions = { }

    def


register_function(self, func):
        self._functions[func.__name__] = func

    def


handle_connection(self, connection):
        try


:
            while


True:
                # Receive a message



                func_name, args, kwargs = pickle.loads(connectio
                # Run the RPC and send a response



                try


:
                    r = self._functions[func_name](*args,**kwarg
                    connection.send(pickle.dumps(r))
                except Exception as


e:
                    connection.send(pickle.dumps(e))
        except EOFError
```

```
:
            pass
```

要使用这个处理例程，需要将其添加到一个消息服务器中。这里我们可以有多种选择，但相较而言multiprocessing库是一种简单的选择。下面是RPC服务器的示例：

```
from multiprocessing.connection import

Listener
from threading import

Thread

def

rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while

True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client
        t.daemon = True
        t.start()
```

```
# Some remote functions

def

add(x, y):
    return

x + y

def

sub(x, y):
    return

x - y

# Register with a handler

handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Run the server

rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')
```

要从远端的客户端中访问这个服务器，需要创建一个相应的RPC代理类来转发请求。示例如下：

```
import pickle


class RPCProxy

:
    def

__init__(self, connection):
        self._connection = connection
    def

__getattr__(self, name):
        def

do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwarg
            result = pickle.loads(self._connection.recv())
            if

isinstance(result, Exception

):
                raise

result
            return

result
        return

 do_rpc
```

要使用这个代理类，只需要用它来包装发送给服务器端的连接即可。示例如下：

```
>>> from multiprocessing.connection import


Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)
5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rpcserver.py", line 37, in do_rpc
    raise


result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

应该指出的是，有许多消息处理层（比如multiprocessing）已经用pickle将数据做了序列化处理。如果是这样的话，可以去掉对pickle.dumps()和pickle.loads()的调用。

11.8.3 讨论

RPCHandler和RPCProxy类的总体思想相对来说都是比较简单的。如果客户端想调用一个远端函数，比如foo(1, 2, z=3)，代理类就创建出一个包含了函数名和参数的元组('foo', (1, 2), {'z':3})。这个元组经pickle序列化处理后通过连接发送出去。这些步骤都是在RPCProxy类__getattr__()方法返回的闭包do_rpc()中执行的。服务器端接收到消息后执行反序列化处理，然后检查函数名是否已经注册过了。如果是注册过的函数，就用给定的参数调用该函数。把得到的结果（或者异常）进行pickle序列化处理然后再发送回去。

我们给出的示例依赖于multiprocessing模块来完成通信。但是，这种方法也适用于任何其他的消息通信系统。比如，如果想在ZeroMQ上实现RPC，只要把连接对象用适当的ZeroMQ socket对象取代即可。

关于pickle的可靠性，需要重点考虑安全方面的问题（因为聪明的黑客可以创建出特定的消息，使得在执行反序列化处理时得以执行任意的函数）。特别是，绝对不能允许非受信任或者非授权的客户端执行RPC操作，我们肯定不希望对互联网上的任何机器都敞开大门。本节提到的技术应该只在位于防火墙之后的内部网络中使用，不要暴露给外部世界。

作为pickle的替代方案，我们可能会考虑使用JSON、XML或一些其他的数据编码来完成序列化操作。比如，如果用json.loads()和json.dumps()来取代本节中的pickle.loads()和pickle.dumps()的话，那么就可以轻松应用JSON编码了。示例如下：

```
# jsonrpcserver.py

import json


class RPCHandler

:
    def

__init__(self):
        self._functions = { }

    def

register_function(self, func):
        self._functions[func.__name__] = func

    def

handle_connection(self, connection):
        try

:
```

```
            while

True:
                # Receive a message



                func_name, args, kwargs = json.loads(connection.r
                # Run the RPC and send a response



                try

:
                    r = self._functions[func_name](*args,**kwargs
                    connection.send(json.dumps(r))
                except Exception as

e:
                    connection.send(json.dumps(str(e)))
        except EOFError

:
            pass




# jsonrpcclient.py



import json




class RPCProxy
```

```
:
    def


__init__(self, connection):
        self._connection = connection
    def


__getattr__(self, name):
        def


do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs
            result = json.loads(self._connection.recv())
            return


result
        return


do_rpc
```

在实现RPC时一个比较复杂的问题是对待异常应该如何处理？最基本的要求是如果调用某个方法时抛出了异常，服务器不会因此而崩溃。但是，如何将异常信息回传给客户端则需要好好思考一番。如果正在使用pickle，通常异常实例会经过序列化处理之后再在客户端重新抛出。如果在使用其他的编码协议，那就要考虑其他的方法了。至少，应该在响应中返回异常信息字符串。上面使用JSON编码的示例采用的正是这种方式。

有关RPC实现的另一个例子，看看在XML-RPC中使用的SimpleXMLRPCServer和ServerProxy类的实现是很有帮助的，我们在11.6节已经描述过了。

11.9　以简单的方式验证客户端身份

11.9.1　问题

我们希望有一种简单的方式可以对在分布式系统中连接到各个服务器上的客户端进行身份验证，但是又不想使用像SSL那样的复杂组件。

11.9.2　解决方案

我们可以利用hmac模块实现一个握手连接来达到简单且高效的身份验证目的。下面是示例代码：

```
import hmac

import os

def

client_authenticate(connection, secret_key):
    '''
```

```
    Authenticate client to a remote service.

    connection represents a network connection.

    secret_key is a key known only to both client/server.

    '''

    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)

def

server_authenticate(connection, secret_key):
    '''

    Request client authentication.

    '''

    message = os.urandom(32)
    connection.send(message)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    response = connection.recv(len(digest))
    return
```

```
hmac.compare_digest(digest,response)
```

总体思路就是在发起连接时，服务器将一段由随机字节组成的消息发送给客户端（在本例中是由os.urandom()返回的）。客户端和服务器通过hmac模块以及双方事先都知道的密钥计算出随机数据的加密hash。客户端发送它计算出的摘要值（digest）给服务器，而服务器对摘要值进行比较，以此来决定是要接受还是拒绝这个连接。

对摘要值进行比较需要使用hmac.compare_digest()函数。这个函数的实现可避免遭受基于时序分析的攻击（timing-analysis attack），因此应该用它来对摘要值进行比较而不能用普通的比较操作符（==）。

要使用这些函数，我们可以将它们合并到已有的有关网络或消息处理的代码中。比如，如果用到了socket，服务器端的代码看起来就是这样的：

```
from socket import

socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'
```

```
def


echo_handler(client_sock):
    if not


server_authenticate(client_sock, secret_key):
        client_sock.close()
        return



    while


True:
        msg = client_sock.recv(8192)
        if not


msg:
            break



        client_sock.sendall(msg)

def


echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while


True:
        c,a = s.accept()
        echo_handler(c)

echo_server(('', 18000))
```

而在客户端，则可以这样处理：

```
from socket import


socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'

s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 18000))
client_authenticate(s, secret_key)
s.send(b'Hello World')
resp = s.recv(1024)
...
```

11.9.3　讨论

在内部消息系统以及进程间通信中常常会用hmac来验证身份。比如，如果正在编写的系统需要实现跨集群的多进程间通信，就可以使用这种方法来确保只有获得许可的进程才能互相通信。事实上，在multiprocessing库中，当同子进程建立通信时在内部也是使用的基于hmac的身份验证方式。

需要重点强调的是，验证某个连接和加密连接可不是一回事。在经过验证的连接上，后续的通信

都是以明文发送的，对于任何企图嗅探流量的人来说，这些消息都是可见的（尽管完成验证所需的密钥从来都没有传送过）。

　　hmac所采用的身份验证算法是基于加密哈希函数的，比如MD5和SHA-1，这些算法的细节描述可以在IETF RFC 2104（http://tools.ietf.org/html/rfc2104.html ）中找到。

11.10　为网络服务增加SSL支持

11.10.1　问题

我们想通过socket实现一个网络服务，要求服务器端和客户端可以通过SSL实现身份验证，并且对传输的数据进行加密。

11.10.2　解决方案

ssl模块可以为底层的socket连接添加对SSL的支持。具体来说就是ssl.wrap_socket()函数可接受一个已有的socket，并为其包装一个SSL层。比方说，下面的示例展示了一个简单的echo服务，服务器对发起连接的客户端提供了一个服务器证书：

```
from socket import


socket, AF_INET, SOCK_STREAM
import ssl



KEYFILE = 'server_key.pem'         # Private key of the server
```

```
CERTFILE = 'server_cert.pem'     # Server certificate (given to c



def


echo_client(s):
    while


True:
        data = s.recv(8192)
        if


data == b'':
            break



        s.send(data)
    s.close()
    print


('Connection closed')

def


echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(1)

    # Wrap with an SSL layer requiring client certs



    s_ssl = ssl.wrap_socket(s,
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            server_side=True
```

```
                            )
    # Wait for connections


    while


True:
        try


:
            c,a = s_ssl.accept()
            print


('Got connection', c, a)
            echo_client(c)
        except Exception as


e:
            print


('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))
```

下面的交互式会话展示了客户端是如何连接到服务器的。客户端要求服务器出示自己的证书并完成验证。

```
>>> from socket import


socket, AF_INET, SOCK_STREAM
```

```
>>> import ssl



>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
...


cert_reqs=ssl.CERT_REQUIRED,
...


ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>
```

这些底层的socket技巧所带来的问题在于，它们无法和已经通过标准库实现的网络服务很好地结合在一起。比方说，大部分的服务器端代码（HTTP、XML-RPC等）实际上是基于socketserver库来实现的。客户端代码也是在更高的层面上来实现的。为已有的服务添加对SSL的支持是可以实现的，但是需要的方法稍有不同。

首先，对于服务器端来说可以通过混入类（mixin class）来添加对SSL的支持：

```
import ssl



class SSLMixin


:
    '''



    Mixin class that adds support for SSL to existing servers ba



    on the socketserver module.



    '''



    def


__init__(self, *args,
                 keyfile=None, certfile=None, ca_certs=None,
                 cert_reqs=ssl.NONE,
                 **kwargs):
        self._keyfile = keyfile
        self._certfile = certfile
        self._ca_certs = ca_certs
        self._cert_reqs = cert_reqs
        super().__init__(*args, **kwargs)

    def


get_request(self):
        client, addr = super().get_request()
        client_ssl = ssl.wrap_socket(client,
                                     keyfile = self._keyfile,
```

```
                                             certfile = self._certfile,
                                             ca_certs = self._ca_certs,
                                             cert_reqs = self._cert_reqs,
                                              server_side = True)
        return

client_ssl, addr
```

要使用这个混入类，可以将它和别的服务器类混合在一起使用。比如，下面的示例定义了一个运行在SSL之上的XML-RPC服务器：

```
# XML-RPC server with SSL

from xmlrpc.server import

SimpleXMLRPCServer

class SSLSimpleXMLRPCServer

(SSLMixin, SimpleXMLRPCServer):
    pass
```

下面的XML-RPC服务器代码取自11.6节，只做

了些许修改以支持SSL：

```
import ssl



from xmlrpc.server import


SimpleXMLRPCServer
from sslmixin import


SSLMixin

class SSLSimpleXMLRPCServer


(SSLMixin, SimpleXMLRPCServer):
    pass




class KeyValueServer


:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def


__init__(self, *args, **kwargs):
        self._data = {}
        self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True
        for


name in


self._rpc_methods_:
```

```
            self._serv.register_function(getattr(self, name))

    def

get(self, name):
        return

self._data[name]

    def

set(self, name, value):
        self._data[name] = value

    def

delete(self, name):
        del

self._data[name]

    def

exists(self, name):
        return

name in

self._data

    def

keys(self):
        return
```

```
list(self._data)

    def


serve_forever(self):
        self._serv.serve_forever()

if


__name__ == '__main__':
    KEYFILE='server_key.pem'     # Private key of the server



    CERTFILE='server_cert.pem'    # Server certificate



    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE)
    kvserv.serve_forever()
```

要使用这个服务器，可以利用xmlrpc.client模块来完成连接。只要在URL中指定一个https:即可。示例如下：

```
>>> from xmlrpc.client import


ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo','bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
```

```
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

SSL客户端中一个比较复杂的问题在于如何执行额外的步骤来验证服务器证书，或者向服务器展示客户端的凭证（比如客户端证书）。不幸的是，似乎并没有标准的方法来完成这些任务，因此常常需要做一点研究。下面的示例展示了如何建立一条安全的XML-RPC连接来验证服务器证书：

```
from xmlrpc.client import


SafeTransport, ServerProxy
import ssl




class VerifyCertSafeTransport


(SafeTransport):
    def


__init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
```

```
        if

cert:
            self._ssl_context.load_cert_chain(certfile, keyfile)
        self._ssl_context.verify_mode = ssl.CERT_REQUIRED

    def

make_connection(self, host):
        # Items in the passed dictionary are passed as keyword

        # arguments to the http.client.HTTPSConnection() construc

        # The context argument allows an ssl.SSLContext instance

        # be passed with information about the SSL configuration

        s = super().make_connection((host, {'context': self._ssl_

        return

s

# Create the client proxy

s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pe
                allow_none=True)
```

我们前面给出的解决方案中是由服务器端向客户端发送证书，然后客户端来验证。其实，这个验证步骤可以在两个方向上进行（即，服务器到客户端、客户端到服务器）。如果服务器想要验证客户端，只要把启动服务器的代码修改为如下形式即可：

```
if

__name__ == '__main__':
    KEYFILE='server_key.pem'      # Private key of the server

    CERTFILE='server_cert.pem'    # Server certificate

    CA_CERTS='client_cert.pem'     # Certificates of accepted cli

    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            ca_certs=CA_CERTS,
                            cert_reqs=ssl.CERT_REQUIRED,
                            )
    kvserv.serve_forever()
```

要让XML-RPC客户端发出自己的证书，需要

把ServerProxy的初始化修改为如下方式：

```
# Create the client proxy



s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pe
                                                  'client_cert.pe
                                                  'client_key.pem
                allow_none=True)
```

11.10.3　讨论

要让本节中提到的技术能正常运转，这对于我们的系统配置能力和对SSL的理解都将是一场考验。也许最大的挑战就是按顺序获取密钥的初始配置、证书以及其他一些相关的细节。

我们来理清一下这里的需求，SSL连接的每个端点一般来说都有一个私有的密钥和一个签名的证书文件。证书文件中包含有公有密钥，在每个连接中会发送给对端节点。对于面向大众的服务，证书一般会由证书授权机构如Verisign、Equifax或者类似的组织（需要付费的机构）来签名。要验证服务器端的证书，客户端会维护一个文件，其中包含有受信任的证书颁发机构所发布的证书。比方说，

Web浏览器维护着与主要的证书颁发机构相对应的证书，并且会在HTTPS连接中用这些证书来验证由Web服务器发送过来的证书的完整性。

为了验证本节提到的技术，可以创建一个被称之为自签名的证书。可以这样来实现：

```
bash % openssl req -new -x509 -days 365 -nodes -out server_cert.p



           -keyout server_key.pem
Generating a 1024 bit RSA private key
..........................................++++++
...++++++
writing new private key to 'server_key.pem'
 -----
You are about to be asked to enter information that will be inco
into your certificate request.
What you are about to enter is what is called a Distinguished Nar
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
 -----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Illinois
Locality Name (eg, city) []:Chicago
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dabea
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:localhost
Email Address []:
bash %
```

当创建证书时，各个字段的值常常是随机的。但是，“Common Name”字段通常会包含DNS服务器

的主机名。如果只是在自己的机器上做下测试，可以用“localhost”来替代。否则就用运行服务器程序的机器域名。

这样配置的结果就是我们将得到一个server_key.pem文件，其中包含了私钥。它看起来是这样的：

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCZrCNLoEyAKF+f9UNcFaz5Osa6jf7qkbUl8si5xQrY3ZYC
nL1dZLn/VbEFIITaUOgvBtPv1qUWTJGwga62VSG1oFE0ODIx3g2Nh4sRf+ry
L4442nx0z4O5vJQ7k6eRNHAZUUnCL50+YvjyLyt7ryLSjSuKhCcJsbZgPwID
AoGAB5evrr7eyL4160tM5rHTeATlaLY3UBOe5Z8XN8Z6gLiB/ucSX9AysviV
3oD6z2aL8jbeJc1vHqjt0dC2dwwm32vVl8mRdyoAsQpWmiqXrkvP4Bsl04Vp
Qt8xNSW9SFhceL3LEvw9M8i9MV39viih1ILyH8OuHdvJyFECQQDLEjl2d2pp
PoLqVFAirDfX2JnLTdWbc+M11a9Jdn3hKF8TcxfEnFVs5Gav1MusicY5KB0y
YbTvqKc7AkEAwbnRBO2VYEZsJZp2X0IZqP9ovWokkpYx+PE4+c6MySDgaMci
WDIHJG1CHudD09GbqENasDzyb2HAIW4CzQJBAKDdkv+xoW6gJx42Auc2WzTc
eXR/+BLpPrhKykzbvOQ8YvS5W764SUO1u1LWs3G+wnRMvrRvlMCZKgggBjkC
Jewto2+a+WkOKQXrNNScCDE5aPTmZQc5waCYq4UmCZQcOjkUOiN3ST1U5iux
V/yX6fw0qh+fLWtkOs/JAkA+okMSxZwqRtfgOFGBfwQ8/iKrnizeanTQ3L6s
CHZXdJ3XQ6qUmNxNn7iJ7S/LDawo1QfWkCfD9FYoxBlg
-----END RSA PRIVATE KEY-----
```

而在server_cert.pem中的服务器端证书看起来也很类似：

```
-----BEGIN CERTIFICATE-----
MIIC+DCCAmGgAwIBAgIJAPMd+vi45js3MA0GCSqGSIb3DQEBBQUAMFwxCzAJ
BAYTAlVTMREwDwYDVQQIEwhJbGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEU
A1UEChMLRGFiZWF6LCBMTEMxEjAQBgNVBAMTCWxvY2FsaG9zdDAeFw0xMzAx
ODQyMjdaFw0xNDAxMTExODQyMjdaMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQI
bGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBM
EjAQBgNVBAMTCWxvY2FsaG9zdDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkC
```

```
mawjS6BMgChfn/VDXBWs+TrGuo3+6pG1JfLIucUK2N2WAu47rpy9XWS5/1Wx
2lDoLwbT79alFkyRsIGutlUhtaBRNDgyMd4NjYeLEX/q8krMdi+OONp8dM+D
O5OnkTRwGVFJwi+dPmL48i8re68i0o0rioQnCbG2YD8CAwEAAaOBwTCBvjAd
HQ4EFgQUrtoLHHgXiDZTr26NMmgKJLJLFtIwgY4GA1UdIwSBhjCBg4AUrtoL
iDZTr26NMmgKJLJLFtKhYKReMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhJ
bm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMx
BgNVBAMTCWxvY2FsaG9zdIIJAPMd+vi45js3MAwGA1UdEwQFMAMBAf8wDQYJ
hvcNAQEFBQADgYEAFci+dqvMG4xF8UTnbGVvZJPIzJDRee6Nbt6AHQo9pOdA
WsGCplSOaDNdKKzl+b2UT2Zp3AIW4Qd51bouSNnR4M/gnr9ZD1ZctFd3jS+C5
D3vvcW5lAnCCC80P6rXy7d7hTeFu5EYKtRGXNvVNd/06NALGDflrrOwxF3Y=
-----END CERTIFICATE-----
```

在与服务器相关的代码中，私钥和证书文件都需要传递给各种SSL相关的包装函数中。证书会呈现给客户端，而私钥则应该受到保护，就保留在服务器端。

在与客户端相关的代码中，我们需要维护一个特殊的文件，其中包含有合法的证书颁发机构信息，以此来验证服务器端的证书。如果没有这样的文件，那么至少可以把服务器端的证书拷贝一份放在客户端机器上，用这个拷贝作为验证的手段。在建立连接时，服务器会发送自己的证书，而我们就可以用已经保存好的证书来完成验证。

服务器也可以选择验证客户端的身份。要做到这一点，客户端需要有自己的私钥和证书。而服务器端也需要维护一个受信任的证书颁发机构信息文件，以此来验证客户端的证书。

如果真的想在网络服务中添加对SSL的支持，本节仅仅只是小试身手告诉你如何完成设置。你肯定需要参考有关文档（http://docs.python.org/3/library/ssl.html）以了解更多的细节。准备好花大量的时间对代码进行试验吧，直到程序能正常工作为止。

11.11　在进程间传递socket文件描述符

11.11.1　问题

我们正在运行多个Python解释器进程，想把一个打开的文件描述符从一个解释器传递到另一个解释器上。例如，也许这里有一个服务器进程负责接收连接，但是实际处理客户端的请求是通过另一个不同的解释器来完成的。

11.11.2　解决方案

要在进程间传递文件描述符，首选需要将进程连接在一起。在UNIX系统上，你可能要用到UNIX域socket，而在Windows上可以使用命名管道。但是，与其同这些底层的进程间通信机制打交道，利用multiprocessing模块来建立这样的连接通常会简单得多。

一旦进程间的连接建立起来了，就可以使用multiprocessing.reduction模块中的send_handle()和recv_handle()函数来在进程之间传送文件描述符了。下面的示例给出了基本用法：

```
import multiprocessing

from multiprocessing.reduction import

recv_handle, send_handle
import socket


def

worker(in_p, out_p):
    out_p.close()
    while

True:
        fd = recv_handle(in_p)
        print

('CHILD: GOT FD', fd)
        with

socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as

s:
            while

True:
                msg = s.recv(1024)
                if not

msg:
                    break
```

```
                print

('CHILD: RECV {!r}'.format(msg))
                s.send(msg)

def

server(address, in_p, out_p, worker_pid):
    in_p.close()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(address)
    s.listen(1)
    while

True:
        client, addr = s.accept()
        print

('SERVER: Got connection from', addr)
        send_handle(out_p, client.fileno(), worker_pid)
        client.close()

if

__name__ == '__main__':
    c1, c2 = multiprocessing.Pipe()
    worker_p = multiprocessing.Process(target=worker, args=(c1,c2
    worker_p.start()

    server_p = multiprocessing.Process(target=server,
                  args=(('', 15000), c1, c2, worker_p.pid))
    server_p.start()

    c1.close()
    c2.close()
```

在这个示例中我们生成了两个进程，并且利用multiprocessing模块的Pipe对象将它们连接在一起。服务器进程打开一个socket并等待客户端的连接。工作者进程只是通过recv_handle()在管道上等待接收文件描述符。当服务器接收到一条连接时，它会将得到的socket文件描述符通过send_handle()发送给工作者进程。工作者进程接管这个socket并将数据回显给客户端直到连接关闭为止。

如果使用Telnet或者类似的工具去连接服务器，那么可能会看到如下的结果：

```
bash % python3 passfd.py
SERVER: Got connection from ('127.0.0.1', 55543)
CHILD: GOT FD 7
CHILD: RECV b'Hello\r\n'
CHILD: RECV b'World\r\n'
```

这个例子中最重要的部分就是服务器端接收到的客户端socket实际上是由另一个进程去处理的。服务器仅仅只是将它转手出去、关闭它然后等待下一个连接。

11.11.3　讨论

有许多程序员甚至都没有意识到在进程之间传递文件描述符是可以实现的。这种技术在构建可扩展的系统时会是一件有用的工具。比如在多核机器上，我们会同时运行多个Python解释器实例，用传递文件描述符的方式来对每个解释器处理的客户端数量做负载均衡。

解决方案中出现的send_handle()和recv_handle()函数只能用于多进程连接的环境中。除了使用管道之外，还可以按照11.7节中介绍的方法来连接解释器，只要你用的是UNIX域socket或者Windows命名管道就可以。比如，可以将服务器和工作者进程实现为完全分离的程序，可以分别启动。下面是服务器端的实现：

```
# servermp.py

from multiprocessing.connection import
Listener
from multiprocessing.reduction import
send_handle
import socket

def
```

```
server(work_address, port):
    # Wait for the worker to connect



    work_serv = Listener(work_address, authkey=b'peekaboo')
    worker = work_serv.accept()
    worker_pid = worker.recv()

    # Now run a TCP/IP server and send clients to worker



    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while


True:
        client, addr = s.accept()
        print


('SERVER: Got connection from', addr)
        send_handle(worker, client.fileno(), worker_pid)
        client.close()

if


__name__ == '__main__':
    import sys



    if


len(sys.argv) != 3:
        print
```

```
('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit


(1)

    server(sys.argv[1], int(sys.argv[2]))
```

要运行这个服务器，可以输入这样的命令：python3 servermp.py /tmp/servconn 15000。下面是对应的客户端代码：

```
# workermp.py




from multiprocessing.connection import


Client
from multiprocessing.reduction import


recv_handle
import os



from socket import


socket, AF_INET, SOCK_STREAM

def
```

```
worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo')
    serv.send(os.getpid())
    while


True:
        fd = recv_handle(serv)
        print


('WORKER: GOT FD', fd)
        with


socket(AF_INET, SOCK_STREAM, fileno=fd) as


client:
            while


True:
                msg = client.recv(1024)
                if not


msg:
                    break



                print


('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if


__name__ == '__main__':
    import sys
```

```
    if

len(sys.argv) != 2:
        print

('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit

(1)

    worker(sys.argv[1])
```

要运行工作者进程，可以输入python3 workermp.py /tmp/servconn。得到的结果应该和使用Pipe()的例子完全一样。

在底层，文件描述符的传递涉及创建UNIX域socket，并且要用到socket的sendmsg()方法。由于这个技术并不是广为人知，下面我们给出另一种不同的服务器实现，展示了如何利用socket来传递文件描述符：

```
# server.py

import socket
```

```
import struct

def

send_fd(sock, fd):
    '''

    Send a single file descriptor.

    '''

    sock.sendmsg([b'x'],
                 [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.
    ack = sock.recv(2)
    assert

ack == b'OK'

def

server(work_address, port):
    # Wait for the worker to connect

    work_serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM
    work_serv.bind(work_address)
    work_serv.listen(1)
    worker, addr = work_serv.accept()

    # Now run a TCP/IP server and send clients to worker
```

```
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('',port))
    s.listen(1)
    while

True:
        client, addr = s.accept()
        print

('SERVER: Got connection from', addr)
        send_fd(worker, client.fileno())
        client.close()

if

__name__ == '__main__':
    import sys

    if

len(sys.argv) != 3:
        print

('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit

(1)

    server(sys.argv[1], int(sys.argv[2]))
```

下面是用socket实现的工作者进程：

```
# worker.py

import socket

import struct


def

recv_fd(sock):
    '''

    Receive a single file descriptor

    '''


    msg, ancdata, flags, addr = sock.recvmsg(1,
                                     socket.CMSG_LEN(struct.calcs

    cmsg_level, cmsg_type, cmsg_data = ancdata[0]
    assert

cmsg_level == socket.SOL_SOCKET and

cmsg_type == socket.SCM_RIGHTS
```

```
    sock.sendall(b'OK')
    return


struct.unpack('i', cmsg_data)[0]

def


worker(server_address):
    serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    serv.connect(server_address)
    while


True:
        fd = recv_fd(serv)
        print


('WORKER: GOT FD', fd)
        with


socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as


client:
            while


True:
                msg = client.recv(1024)
                if not


msg:
                    break



                print


('WORKER: RECV {!r}'.format(msg))
```

```
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)
    worker(sys.argv[1])
```

如果打算在自己的程序中传递文件描述符，那么读一些相关的高级材料比如W.Richard Stevens所著的*Unix Network Programming*（Prentice Hall, 1990）是非常明智的。在Windows上传递文件描述符需要使用与UNIX不同的技术（本节未给出）。对于Windows平台，建议学习一下multiprocessing.reduction的源码，研究其中的细节来看看到底是如何实现的。

11.12　理解事件驱动型I/O

11.12.1　问题

我们可能已经听说过某些Python的包是基于“事件驱动”或“异步”I/O的，但是并不能完全理解这到底是什么意思，在底层它究竟是如何工作的，或者说如果使用了这样的技术会对程序产生什么影响。

11.12.2　解决方案

从根本上说，事件驱动I/O是一种将基本的I/O操作（即，读和写）转换成事件的技术，而我们必须在程序中去处理这种事件。比方说，当在socket上收到数据时，这就成为一个“接收事件”，由我们提供的回调方法或者函数负责处理以此来响应这个事件。一个事件驱动型框架可能会以一个基类作为起始点，实现一系列基本的事件处理方法，就像下面的示例这样：

```
class EventHandler

:
```

```
    def


fileno(self):
        'Return the associated file descriptor'
        raise


NotImplemented('must implement')

    def


wants_to_receive(self):
        'Return True if receiving is allowed'
        return


False

    def


handle_receive(self):
        'Perform the receive operation'
        pass




    def


wants_to_send(self):
        'Return True if sending is requested'
        return


False

    def


handle_send(self):
        'Send outgoing data'
```

```
        pass
```

之后，就可以把这个类的实例插入到一个事件循环中，看起来是这样的：

```
import select



def


event_loop(handlers):
    while


True:
        wants_recv = [h for


h in


handlers if


h.wants_to_receive()]
        wants_send = [h for


h in


handlers if
```

```
h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv, wants_s
        for


h in


can_recv:
            h.handle_receive()
        for


h in


can_send:
            h.handle_send()
```

就这么简单！事件循环的核心在于select()调用，它会轮询文件描述符检查它们是否处于活跃状态。在调用select()之前，事件循环会简单地查询所有的处理方法，看它们是希望接收还是发送数据。然后把查询的结果以列表的方式提供给select()。结果就是，select()会返回已经在接收或发送事件上就绪的对象列表。对应的handle_receive()或者handle_send()方法就被触发执行。

要编写应用程序，就需要创建特定的EventHandler类的实例。比如，这里有两个简单的处理程序，用以说明两个基于UDP的网络服务：

```
import socket

import time

class UDPServer

(EventHandler):
    def

__init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGR
        self.sock.bind(address)

    def

fileno(self):
        return

self.sock.fileno()

    def

wants_to_receive(self):
        return

True

class UDPTimeServer

(UDPServer):
    def
```

```
handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer


(UDPServer):
    def


handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)

if


__name__ == '__main__':
    handlers = [ UDPTimeServer(('',14000)), UDPEchoServer(('',150
    event_loop(handlers)
```

要测试这份代码，可以试着从另一个Python解释器中连接服务器：

```
>>> from socket import


*
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'',('localhost',14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello',('localhost',15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
```

```
>>>
```

实现一个TCP服务器就要稍微复杂一些，因为每个客户端都涉及产生一个新的处理对象。下面是TCP echo客户端的示例：

```
class TCPServer


(EventHandler):
    def


__init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STR
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEAD
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def


fileno(self):
        return


self.sock.fileno()

    def


wants_to_receive(self):
        return
```

```
True

    def


handle_receive(self):
        client, addr = self.sock.accept()
        # Add the client to the event loop's handler list



        self.handler_list.append(self.client_handler(client, sel
class TCPClient


(EventHandler):
    def


__init__(self, sock, handler_list):
        self.sock = sock
        self.handler_list = handler_list
        self.outgoing = bytearray()

    def


fileno(self):
        return


self.sock.fileno()

    def


close(self):
        self.sock.close()
        # Remove myself from the event loop's handler list



        self.handler_list.remove(self)

    def
```

```
wants_to_send(self):
        return


True if


self.outgoing else


False

    def


handle_send(self):
        nsent = self.sock.send(self.outgoing)
        self.outgoing = self.outgoing[nsent:]

class TCPEchoClient


(TCPClient):
    def


wants_to_receive(self):
        return


True

    def


handle_receive(self):
        data = self.sock.recv(8192)
        if not


data:
            self.close()
        else
```

```
:
            self.outgoing.extend(data)

if


__name__ == '__main__':
    handlers = []
    handlers.append(TCPServer(('',16000), TCPEchoClient, handlers
    event_loop(handlers)
```

这个TCP示例的关键在于需要从处理列表中添加和移除客户端。在每个连接中都会为客户端创建一个新的处理例程并添加到列表中。可是当连接关闭时，每个客户端都必须将它们自己从列表中移除出去。

如果运行这个程序并尝试用Telnet或者类似的工具来建立连接，就会看到服务器会将接收到的数据回送给你。这个程序应该能轻松处理多个不同的客户端。

11.12.3　讨论

几乎所有的事件驱动型框架的工作原理都和我们给出的解决方案相类似。实际的实现细节以及软件的总体架构可能会有较大的区别，但是核心部分

都有一个循环来轮询socket的活跃性并执行响应操作。

事件驱动型I/O的一个潜在优势在于它可以在不使用线程和进程的条件下同时处理大量的连接。也就是说，select()调用（或者功能相同的其他调用）可用来监视成百上千个socket，并且针对它们中间发生的事件作出响应。事件循环一次处理一个事件，不需要任何其他的并发原语参与。

事件驱动型I/O的缺点在于这里并没有涉及真正的并发。如果任何一个事件处理方法阻塞了或者执行了一个耗时较长的计算，那么就会阻塞整个程序的执行进程。不是以事件驱动风格实现的库函数调用起来也会有这个问题。总是会有这样的风险，即，某些库函数调用阻塞了，导致整个事件循环停滞不前。

对于阻塞型或者需要长时间运行的计算，可以通过将任务发送给单独的线程或者进程来解决。但是，将线程和进程同事件循环进行协调需要较高的技巧。下面的代码示例通过concurrent.futures模块来实现：

```
from concurrent.futures import
```

```
ThreadPoolExecutor
import os




class ThreadPoolHandler


(EventHandler):
    def


__init__(self, nworkers):
        if


os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socke
        else


:
            server = socket.socket(socket.AF_INET, socket.SOCK_S
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                  socket.SOCK_ST
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def


fileno(self):
        return


self.done_sock.fileno()

    # Callback that executes when the thread is done
```

```
    def

_complete(self, callback, r):
        self.pending.append((callback, r.result()))
        self.signal_done_sock.send(b'x')

    # Run a function in a thread pool

    def

run(self, func, args=(), kwargs={},*,callback):
        r = self.pool.submit(func, *args, **kwargs)
        r.add_done_callback(lambda

r: self._complete(callback, r))

    def

wants_to_receive(self):
        return

True

    # Run callback functions of completed work

    def

handle_receive(self):
        # Invoke all pending callback functions
```

```
        for

callback, result in

self.pending:
            callback(result)
            self.done_sock.recv(1)
        self.pending = []
```

在这份代码中，run()方法用来将任务以及任务完成时需要触发的回调函数一起提交到线程池中。实际的任务就提交给了ThredPoolExecutor实例。但是，一个非常棘手的地方在于需要考虑计算出的结果同事件循环之间的协调同步问题。为了实现这个目的，我们在底层创建了一对socket，用来实现一种信号通知机制。当线程池中的任务完成后，它就执行类中的_complete()方法。该方法在对这些socket写入一字节的数据前，将暂停的回调函数和结果进行排队处理。fileno()方法可用来返回另一个socket。因此，当写入这个字节时就会通知事件循环有事件发生了。当触发时，handle_receive()方法将执行所有之前提交过来的回调函数。坦白说，这足以把人的脑袋弄晕。

下面给出了一个简单的服务器实现，展示了如何利用线程池来执行需要长时间运行的计算任务：

```
# A really bad Fibonacci implementation



def


fib(n):
    if


n < 2:
        return


1
    else


:
        return


fib(n - 1) + fib(n - 2)

class UDPFibServer


(UDPServer):
    def


handle_receive(self):
        msg, addr = self.sock.recvfrom(128)
        n = int(msg)
        pool.run(fib, (n,), callback=lambda


r: self.respond(r, addr))

    def


respond(self, result, addr):
```

```
        self.sock.sendto(str(result).encode('ascii'), addr)
if


__name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [ pool, UDPFibServer(('',16000))]
    event_loop(handlers)
```

要测试这个服务器，只需要运行上面的代码并通过另一个Python程序来做些试验：

```
from socket import


*
sock = socket(AF_INET, SOCK_DGRAM)
for


x in


range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
    print


(resp[0])
```

我们应该可以在许多不同的窗口中重复运行这个程序，这么做不会使其他的程序被阻塞。但是我

们运行的程序实例越多，它运行的速度也会越来越慢。

读完这一节后你会考虑使用这样的代码吗？很可能不会。相反，应该找一个功能完备的框架来完成同样的任务。但是，如果理解了本节提到的基本概念，就能理解这样的框架所采用的核心技术。作为基于回调的编程模型的替代方案，有时候也会在事件驱动型代码中使用协程。请参见12.12节中的示例。

11.13　发送和接收大型数组

11.13.1　问题

我们想通过网络连接发送和接收大型数组，其中的数据都是连续的，并且要求尽可能少地对数据进行拷贝。

11.13.2　解决方案

下面的函数利用memoryview对大型数组进行发送和接收：

```
# zerocopy.py

def

send_from(arr, dest):
    view = memoryview(arr).cast('B')
    while

len(view):
        nsent = dest.send(view)
        view = view[nsent:]

def
```

```
recv_into(arr, source):
    view = memoryview(arr).cast('B')
    while


len(view):
        nrecv = source.recv_into(view)
        view = view[nrecv:]
```

要测试这个程序，首先创建一个服务器和客户端程序，它们之间通过socket进行连接。服务器端的代码如下：

```
>>> from socket import


*
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
>>>
```

客户端代码如下（在另一个单独的解释器中运行）：

```
>>> from socket import


*
```

```
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

现在来看看本节所关注的主要问题：可以在网络连接上传输大型的数组。这种情况下，数组可以通过array模块或者numpy来创建。示例如下：

```
# Server
>>> import numpy



>>> a = numpy.arange(0.0, 50000000.0)
>>> send_from(a, c)
>>>

# Client
>>> import numpy



>>> a = numpy.zeros(shape=50000000, dtype=float)
>>> a[0:10]
array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> recv_into(a, c)
>>> a[0:10]
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>>
```

11.13.3　讨论

在数据密集型的分布式计算以及采用并行编程技术的应用程序中，编写需要发送和接收大型数据块的程序是很常见的。但是为了实现这个目标，需要以某种方式将数据还原为原始的字节给底层的网络接口所用。我们可能还需要将数据分片为较小的块，因为大部分与网络相关的函数都无法一次性发送或者接收超大型的数据块。

一种方法是以某种方式将数据进行序列化处理——可能是将其转换为字节串的形式。但是，这么做通常都要对数据进行拷贝，这正是我们极力避免的。就算我们将数据逐块进行拷贝，代码中还是会产生大量的小块拷贝操作。

本节提到的技术对这个问题进行了规避，这是通过利用memoryview来实现的一个技巧。从本质上来说，memoryview就是对已有数组的一层覆盖。不仅是这样，memoryview还可以转型为不同的类型，允许数据根据不同的方式进行解释。这正是下列语句的用意所在：

```
view = memoryview(arr).cast('B')
```

上面的语句接受一个数组arr，并将其转型为无符号字节的memoryview。

这种形式的memoryview可以传递给与socket相关的函数，比如sock.send()或者send.recv_into()。在底层，这些方法可以直接同内存打交道。比如，sock.send()直接从内存中发送数据，不需要进行拷贝。针对接收操作，send.recv_into()会将memoryview作为输入缓冲区来使用。

避免内存拷贝的问题解决了，剩下的问题就主要归结在与socket相关的函数一次只能处理一部分数据上。一般来说，需要调用多次send()和recv_into()才能将整个数组传输完毕。别担心，每次操作后，memoryview都会根据已经发送或者接收的字节数做切片处理，以产生一个新的memoryview。这个新的memoryview同样也是一个内存覆盖层，因此根本不会产生任何拷贝。

这里还有一个问题就是接收方需要预先知道要对方要发送多少数据，这样接收方才可以预分配一个相应的数组，或者验证是否可以将接收到的数据直接放入已有的数组中。如果这对你来说存在问题，那么可以让发送方总是先发送数据的大小，后面再跟着数组数据。

第12章　并发

Python很早就开始支持多种不同的并发编程方法，包括多线程、加载子进程以及各种涉及生成器函数的技巧。在本章中，我们会谈到有关并发编程的方方面面，包括常见的多线程编程技术以及实现并行处理的方法。

有经验的程序员都应该知道，并发编程中充满了潜在的危险。因此，本章的重点在于引导大家编写出更可靠以及更易于调试的代码。

12.1 启动和停止线程

12.1.1 问题

为了让代码能够并发执行，我们想创建线程并在合适的时候销毁。

12.1.2 解决方案

threading库可用来在单独的线程中执行任意的Python可调用对象。要实现这一要求，可以创建一个Thread实例并为它提供期望执行的可调用对象。下面是一个简单的示例：

```
# Code to execute in an independent thread

import time

def

countdown(n):
    while

n > 0:
        print
```

```
('T-minus', n)

n -= 1

time.sleep(5)
# Create and launch a thread

from threading import

Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

当创建一个线程实例时，在调用它的start()方法之前（需要提供目标函数以及相应的参数），线程并不会立刻开始执行。

线程实例会在它们自己所属的系统级线程（即， POSIX线程或Windows线程）中执行，这些线程完全由操作系统来管理。一旦启动后，线程就开始独立地运行，直到目标函数返回为止。可以查询线程实例来判断它是否还在运行：

```
if

t.is_alive():
    print

('Still running')
else

:
    print

('Completed')
```

也可以请求连接（join）到某个线程上，这么做会等待该线程结束：

```
t.join()
```

解释器会一直保持运行，直到所有的线程都终结为止。对于需要长时间运行的线程或者一直不断运行的后台任务，应该考虑将这些线程设置为daemon（即，守护线程）。示例如下：

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

daemon线程是无法被连接的。但是，当主线程结束后它们会自动销毁掉。

除了以上展示的两种操作外，对于线程没有太多别的操作可做了。比如说，终止线程、给线程发信号、调整线程调度属性以及执行任何其他的高级操作，这些功能都没有。如果想要这些功能，就需要自己去构建。

如果想要终止线程，这个线程必须要能够在某个指定的点上轮询退出状态，这就需要编程实现。比如，可以将线程放到下面这样的类中：

```
class CountdownTask

:
    def

__init__(self):

self._running = True
```

```
    def

terminate(self):

self._running = False
    def

run(self, n):
        while

self._running and

n > 0:
            print

('T-minus', n)

n -= 1

time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
...
c.terminate()     # Signal termination

t.join()         # Wait for actual termination (if needed)
```

如果线程会执行阻塞性的操作比如I/O，那么在轮询线程的退出状态时如何实现同步将变得很棘手。比如，某个线程被永远阻塞在I/O操作上了，那么它就永远无法返回，以检查自己是否要被终止。要正确处理这个问题，需要小心地为线程加上超时循环。示例如下：

```
class IOTask

:
    def

terminate(self):


self._running = False
    def

run(self, sock):


# sock is a socket
```

```
sock.settimeout(5)          # Set timeout period

        while

self._running:

# Perform a blocking I/O operation w/ timeout

            try

:

data = sock.recv(8192)
                break

            except

socket.timeout:
                continue

# Continued processing
```

```
...

# Terminated

        return
```

12.1.3 讨论

由于全局解释器锁（GIL）的存在，Python线程的执行模型被限制为在任意时刻只允许在解释器中运行一个线程。基于这个原因，不应该使用Python线程来处理计算密集型的任务，因为在这种任务中我们希望在多个CPU核心上实现并行处理。Python线程更适合于I/O处理以及涉及阻塞操作的并发执行任务（即，等待I/O、等待从数据库中取出结果等）。

有时候我们会发现从Thread类中继承而来的线程类。比如：

```
from threading import
```

```
Thread

class CountdownThread


(Thread):
    def


__init__(self, n):



super().__init__()



self.n = 0
    def


run(self):
        while


self.n > 0:
            print


('T-minus', self.n)



self.n -= 1



time.sleep(5)

c = CountdownThread(5)
c.start()
```

尽管这么做也能完成任务，但这在代码和threading库之间引入了一层额外的依赖关系。意思就是说，上面的代码只能用在有关线程的上下文中，而我们之前展示的技术中编写的代码并不会显式依赖于threading库。把代码从这种依赖关系中解放出来，那么就可以使代码在其他可能不会涉及线程的上下文中也能得到重用。比如，我们可能会利用multiprocessing模块让代码在单独的进程中运行，代码看起来是这样的：

```
import multiprocessing



c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
...
```

再一次申明，这只会在CountdownTask类独立于任何一种并发机制（线程、进程等）时才有用。

12.2　判断线程是否已经启动

12.2.1　问题

我们已经加载了一个线程，但是想知道它实际会在什么时候开始运行。

12.2.2　解决方案

线程的核心特征就是它们能够以非确定性的方式（即，何时开始执行、何时被打断、何时恢复执行完全由操作系统来调度管理，这是用户和程序员都无法确定的）独立执行。如果程序中有其他线程需要判断某个线程是否已经到达执行过程中的某个点，根据这个判断来执行后续的操作，那么这就产生了非常棘手的线程同步问题。要解决这类问题，我们可以使用threading库中的Event对象。

Event对象和条件标记（sticky flag）类似，允许线程等待某个事件发生。初始状态时事件被设置为0。如果事件没有被设置而线程正在等待该事件，那么线程就会被阻塞（即，进入休眠状态），直到事件被设置为止。当有线程设置了这个事件时，这会唤醒所有正在等待该事件的线程（如果有

的话）。如果线程等待的事件已经设置了，那么线程会继续执行。

下面给出了一个简单的示例，使用Event来同步线程的启动：

```
from threading import

Thread, Event
import time



# Code to execute in an independent thread


def


countdown(n, started_evt):
    print


('countdown starting')



started_evt.set()
    while


n > 0:
        print


('T-minus', n)
```

```
n -= 1

time.sleep(5)
# Create the event object that will be used to signal startup

started_evt = Event()
# Launch the thread and pass the startup event

print

('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()
# Wait for the thread to start

started_evt.wait()
print

('countdown is running')
```

当运行这段代码时，字符串“countdown is running”总是会在“countdown starting”之后显示。这里使用了事件来同步线程，使得主线程等待，直到

countdown()函数首先打印出启动信息之后才开始执行。

12.2.3　讨论

Event对象最好只用于一次性的事件。也就是说，我们创建一个事件，让线程等待事件被设置，然后一旦完成了设置，Event对象就被丢弃。尽管可以使用Event对象的clear()方法来清除事件，但是要安全地清除事件并等待它被再次设置这个过程很难同步协调，可能会造成事件丢失、死锁或者其他的问题（特别是，在设定完事件之后，我们无法保证发起的事件清除请求就一定会在线程再次等待该事件之前被执行）[1] 。

如果线程打算一遍又一遍地重复通知某个事件，那最好使用Condition对象来处理。比如，下面的代码实现了一个周期性的定时器，每当定时器超时时，其他的线程都可以感知到超时事件：

```
import threading

import time
```

```
class PeriodicTimer

:
    def

__init__(self, interval):


self._interval = interval


self._flag = 0


self._cv = threading.Condition()
    def

start(self):


t = threading.Thread(target=self.run)


t.daemon = True


t.start()
    def

run(self):
```

```
'''
Run the timer and notify waiting threads after each interval
'''
        while
True:
time.sleep(self._interval)
            with
self._cv:
self._flag ^= 1
self._cv.notify_all()
def
wait_for_tick(self):
```

```
'''
Wait for the next tick of the timer
'''
    with
self._cv:
last_flag = self._flag
        while
last_flag == self._flag:
self._cv.wait()
# Example use of the timer
ptimer = PeriodicTimer(5)
ptimer.start()
# Two threads that synchronize on the timer
```

```
def

countdown(nticks):
    while

nticks > 0:

ptimer.wait_for_tick()
        print

('T-minus', nticks)

nticks -= 1

def

countup(last):

n = 0
    while

n < last:

ptimer.wait_for_tick()
        print

('Counting', n)
```

```
n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()
```

Event对象的关键特性就是它会唤醒所有等待的线程。如果我们编写的程序只希望唤醒一个单独的等待线程，那么最好使用Semaphore或者Condition对象。

比方说，考虑下面使用了信号量（semaphore）的代码：

```
# Worker thread

def

worker(n, sema):

# Wait to be signaled

sema.acquire()
```

```
# Do some work

    print

('Working', n)

# Create some threads



sema = threading.Semaphore(0)
nworkers = 10
for

n in

range(nworkers):

t = threading.Thread(target=worker, args=(n, sema,))

t.start()
```

执行上面的程序会启动一系列的线程，但是什么也不会发生。这些线程都会因为等待获取信号量而被阻塞。每次释放信号量时，只有一个工作者线程会被唤醒并投入运行。示例如下：

```
>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>
```

如果编写的代码中涉及许多线程间同步的技巧，那么很容易就会让自己的脑袋转晕。一个更为明智的做法是利用队列或者actor模式来完成线程间的通信任务。队列将在下一节中描述。actor模式将在12.10节中讲解。

12.3 线程间通信

12.3.1 问题

我们的程序中有多个线程，我们想在这些线程之间实现安全的通信或者交换数据。

12.3.2 解决方案

也许将数据从一个线程发往另一个线程最安全的做法就是使用queue模块中的Queue（队列）了。要做到这些，首先创建一个Queue实例，它会被所有的线程共享。之后线程可以使用put()或者get()操作来给队列添加或移除元素。示例如下：

```
from queue import


Queue
from threading import


Thread

# A thread that produces data



def
```

```
producer(out_q):
    while

True:

# Produce some data

...

out_q.put(data)

# A thread that consumes data

    def

consumer(in_q):
        while

True:

# Get some data
```

```
data = in_q.get()

# Process the data

...
# Create the shared queue and launch both threads

q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Queue实例已经拥有了所有所需的锁，因此它们可以安全地在任意多的线程之间共享。

当使用队列时，如何对生产者（producer）和消费者（consumer）的关闭过程进行同步协调需要用到一些技巧。这个问题的一般解决方法是使用一个特殊的终止值，当我们将它放入队列中时就使消费者退出。示例如下：

```
from queue import

Queue
from threading import

Thread

# Object that signals shutdown

_sentinel = object()

# A thread that produces data

def

producer(out_q):
    while

running:

# Produce some data

...

out_q.put(data)
```

```
# Put the sentinel on the queue to indicate completion




out_q.put(_sentinel)

# A thread that consumes data


def


consumer(in_q):
    while


True:



# Get some data





data = in_q.get()




# Check for termination



        if


data is
```

```
_sentinel:

in_q.put(_sentinel)
            break

# Process the data

        ...
```

这个示例中有一个很微妙的功能，那就是当消费者接收到这个特殊的终止值后，会立刻将其重新放回到队列中。这么做使得在同一个队列上监听的其他消费者线程也能接收到终止值——因此可以一个一个地将它们都关闭掉。

尽管队列是线程间通信的最常见的机制，但是只要添加了所需的锁和同步功能，就可以构建自己的线程安全型的数据结构。最常见的做法是将你的数据结构和条件变量打包在一起。比如，下面的示例构建了一个线程安全的优先级队列。关于优先级队列我们在1.5节中已经讨论过。

```
import heapq

import threading

class PriorityQueue
:
    def
__init__(self):

self._queue = []

self._count = 0

self._cv = threading.Condition()
    def
put(self, item, priority):
        with
self._cv:

heapq.heappush(self._queue, (-priority, self._count, item))
```

```
self._count += 1

self._cv.notify()

def

get(self):
    with

self._cv:
        while

len(self._queue) == 0:

self._cv.wait()
        return

heapq.heappop(self._queue)[-1]
```

通过队列实现的线程间通信是一种单方向且不确定的过程。一般来说，我们无法得知接收线程（也就是消费者）何时会实际接收到消息并开始工作。但是，Queue对象的确提供了一些基本的事件完成功能（completion feature）。下面的示例通过task_done()和join()方法对此进行了说明：

```
from queue import
```

```
Queue
from threading import

Thread

# A thread that produces data

def

producer(out_q):
    while

running:

# Produce some data

...

out_q.put(data)

# A thread that consumes data

def

consumer(in_q):
    while
```

```
True:

# Get some data

data = in_q.get()

# Process the data

...

# Indicate completion

in_q.task_done()

# Create the shared queue and launch both threads

q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
```

```
t2.start()

# Wait for all produced items to be consumed



q.join()
```

当消费者线程已经处理了某项特定的数据，而生产者线程需要对此立刻感知的话，那么就应该将发送的数据和一个Event对象配对在一起，这样就允许生产者线程可以监视这一过程。示例如下：

```
from queue import


Queue
from threading import


Thread, Event

# A thread that produces data



def


producer(out_q):
    while


running:
```

```
# Produce some data
...
# Make an (data, event) pair and hand it to the consumer
evt = Event()
out_q.put((data, evt))
...
# Wait for the consumer to process the item
evt.wait()
# A thread that consumes data
def
```

```
consumer(in_q):
    while

True:


# Get some data



data, evt = in_q.get()


# Process the data



...


# Indicate completion



evt.set()
```

12.3.3 讨论

把多线程程序按照简单的队列机制来实现，这对于保持程序的清晰性而言通常是一种不错的方式。如果可以把所有的任务都分解成用简单的线程安全型队列来处理，就会发现不需要用锁和其他的底层同步原语把程序弄得一团糟了。此外，使用队列进行通信常常使得程序的设计可以在稍后扩展到其他类型的基于消息通信的模式上。例如，可以将程序分解为多个进程，甚至做成分布式系统，而这一切都不需要对底层基于队列的架构做大的改动。

一个值得注意的地方是，在线程中使用队列时，将某个数据放入队列并不会产生该数据的拷贝。因此，通信过程实际上涉及在不同的线程间传递对象的引用。如果需要关心共享状态，那么只传递不可变的数据结构（即，整数、字符串或者元组），要么就对排队的数据做深拷贝，这就显得合情合理了。示例如下：

```
from queue import

Queue
from threading import

Thread
import copy
```

```
# A thread that produces data

def

producer(out_q):
    while

True:

# Produce some data

...

out_q.put(copy.deepcopy(data))

# A thread that consumes data

def

consumer(in_q):
    while

True:
```

```
# Get some data

data = in_q.get()

# Process the data

        ...
```

Queue对象提供的一些额外功能被证明在特定的上下文中是有帮助的。如果通过一个可选的大小参数来创建Queue对象，例如Queue(N)，那么这就在put()操作阻塞生产者线程之前对可以入队列的元素个数进行了限制。如果在生产者产生数据和消费者处理数据的速度上存在差异时，给队列可容纳的元素个数设定一个上限值就显得很有意义了。例如，如果生产者产生数据的速度比消费数据的速度快得多时。另一方面，当队列满时将其阻塞同样也会在程序中产生意外的连锁效应，可能导致出现死锁或者运行效率低下。总的来说，线程间通信的控制流是一个看似简单实则困难的问题。如果曾经发现自己试图通过调整队列的大小来修正问题，那么

这就表明程序的设计不够健壮或者存在固有的扩展问题。

get()和put()方法都支持非阻塞和超时机制。示例如下:

```
import queue



q = queue.Queue()

try


:



data = q.get(block=False)
except


queue.Empty:



...
try


:



q.put(item, block=False)
except


queue.Full:
```

```
...
try
:
data = q.get(timeout=5.0)
except
queue.Empty:
...
```

这两种机制都可用来避免在特定的队列操作上无限期阻塞下去的问题。比如，可以用非阻塞的put()配合固定大小的队列来实现当队列满时不同类型的处理方法。比如，可以生成一条日志信息然后将数据丢弃：

```
def
producer(q):
...
```

```
    try

:



q.put(item, block=False)
    except


queue.Full:



log.warning('queued item %r discarded!', item)
```

如果想让消费者线程周期性地放弃q.get()这样的操作，以便于它们检查类似结束标记（termination flag，见12.1节）这样的情况，那么超时机制是很有用的。

```
_running = True

def


consumer(q):
    while


_running:
        try


:
```

```
item = q.get(timeout=5.0)

# Process item

...
        except

queue.Empty:
            pass
```

最后，这里还有一些很实用的方法，比如q.qsize()、q.full()、q.empty()，它们能够告诉我们队列的当前大小和状态。但是，请注意所有这些方法在多线程环境中都是不可靠的。例如，对q.empty()的调用可能会告诉我们队列是空的，但是在完成这个调用的同时，另一个线程可能已经往队列中添加了一个元素。坦白讲，在编写代码时最好不要依赖这些函数。

12.4　对临界区加锁

12.4.1　问题

我们的程序用到了多线程，我们想对临界区进行加锁处理以避免出现竞态条件（race condition）。

12.4.2　解决方案

要想让可变对象安全地用在多线程环境中，可以利用threading库中的Lock对象来解决，示例如下：

```
import threading

class SharedCounter
:
'''
```

```
A counter object that can be shared by multiple threads.

'''

    def
__init__(self, initial_value = 0):

self._value = initial_value

self._value_lock = threading.Lock()
    def
incr(self,delta=1):

'''

Increment the counter with locking
```

```
'''

        with

self._value_lock:

self._value += delta
    def

decr(self,delta=1):

'''



Decrement the counter with locking



'''

        with

self._value_lock:
```

```
self._value -= delta
```

当使用with语句时，Lock对象可确保产生互斥的行为——也就是说，同一时间只允许一个线程执行with语句块中的代码。with语句会在执行缩进的代码块时获取到锁，当控制流离开缩进的语句块时释放这个锁。

12.4.3 讨论

线程的调度从本质上来说是非确定性的。正因为如此，在多线程程序中如果不用好锁就会使得数据被随机地破坏掉，以及产生我们称之为竞态条件的奇怪行为。要避免这些问题，只要共享的可变状态需要被多个线程访问，那么就得使用锁。

在比较老的Python代码中，我们常会看到显式地获取和释放锁的动作。例如，对上面的例子稍作修改：

```
import threading
```

```
class SharedCounter

:

'''



A counter object that can be shared by multiple threads.



'''

    def

__init__(self, initial_value = 0):

self._value = initial_value

self._value_lock = threading.Lock()
    def

incr(self,delta=1):
```

```
'''
Increment the counter with locking
'''
self._value_lock.acquire()
self._value += delta
self._value_lock.release()
    def
decr(self,delta=1):
'''
```

```
Decrement the counter with locking

'''

self._value_lock.acquire()

self._value -= delta

self._value_lock.release()
```

采用with语句会更加优雅，也不容易出错——尤其是如果程序刚好在持有锁的时候抛出了异常，而程序员可能忘记去调用release()方法时更是如此（在这两种情况下，with语句会确保总是释放锁）。

要避免可能出现的死锁，用到了锁的程序应该以这样的方式来编写，即，每个线程一次只允许获取一把锁。如果无法做到，我们可能需要在程序中引入更为高级的避免死锁的技术，我们将在12.5节

中进一步讨论。

在threading库中我们会发现还有其他的同步原语，比如RLock和Semaphore对象。一般来说，这些对象都有特殊的用途，不应该用这些对象对可变状态做简单的加锁处理。RLock被称为可重入锁，它可以被同一个线程多次获取，主要用来编写基于锁的代码，或者基于“监视器”的同步处理。当某个类持有这种类型的锁时，只有一个线程可以使用类中的全部函数或者方法。例如，可以将SharedCounter类实现为如下形式：

```
import threading



class SharedCounter

:


'''




A counter object that can be shared by multiple threads.


```

```
'''

_lock = threading.RLock()
    def

__init__(self, initial_value = 0):

self._value = initial_value

    def

incr(self,delta=1):

'''

Increment the counter with locking

'''
```

```
        with
SharedCounter._lock:
self._value += delta
    def
decr(self,delta=1):
'''
Decrement the counter with locking
'''
        with
SharedCounter._lock:
self.incr(-delta)
```

这份代码中只有一个作用于整个类的锁，它被所有的类实例所共享。不再将锁绑定到每个实例的可变状态上，现在这个锁是用来同步类中的方法的。具体来说，这个锁可确保每次只有一个线程可以使用类中的方法。但是和标准的锁不同的地方在于，对于已经持有了锁的方法可以调用同样使用了这个锁的其他方法（参考decr()方法的实现）。

这个实现的特点之一是无论创建了多少个counter实例，都只会有一个锁存在。因此，当有大量counter对象存在时，这种方法对内存的使用效率要高很多。但是，可能存在的缺点是在使用了大量线程且需要频繁更新counter的程序中，这么做会产生更多的锁争用问题。

Semaphore对象是一种基于共享计数器的同步原语。如果计数器非零，那么with语句会递减计数器并且允许线程继续执行。当with语句块结束时计数器会得到递增。如果计数器为零，那么执行过程会被阻塞，直到由另一个线程来递增计数器为止。尽管Semaphore可以和标准的Lock对象一样以相同的方式来使用，但是由于Semaphore的实现更为复杂，这会对程序的性能带来负面影响。除了简单的加锁功能之外，Semaphore对象对于那些涉及在线程间发送信号或者需要实现节流（throttling）处理的应用中更加有用。例如，如果想在代码中限制并

发的总数，可以使用Semaphore来处理：

```
from threading import


Semaphore
import urllib.request




# At most, five threads allowed to run at once



_fetch_url_sema = Semaphore(5)

def


fetch_url(url):
    with


_fetch_url_sema:
        return


urllib.request.urlopen(url)
```

如果对于线程同步原语背后的理论和实现感兴趣的话，可以参考几乎任何一本有关操作系统的教科书。

12.5 避免死锁

12.5.1 问题

我们正在编写一个多线程程序，线程一次需要获取不止一把锁，同时还要避免出现死锁。

12.5.2 解决方案

在多线程程序中，出现死锁的常见原因就是线程一次尝试获取了多个锁。例如，如果有一个线程获取到第一个锁，但是在尝试获取第二个锁时阻塞了，那么这个线程就有可能会阻塞住其他线程的执行，进而使得整个程序僵死。

避免出现死锁的一种解决方案就是给程序中的每个锁分配一个唯一的数字编号，并且在获取多个锁时只按照编号的升序方式来获取。利用上下文管理器来实现这个机制非常简单，示例如下：

```
import threading

from contextlib import
```

```
contextmanager

# Thread-local state to stored information on locks already acqu

_local = threading.local()

@contextmanager
def

acquire(*locks):

# Sort locks by object identifier

locks = sorted(locks, key=lambda

x: id(x))

# Make sure lock order of previously acquired locks is not viola

acquired = getattr(_local,'acquired',[])
    if

acquired and
```

```
max(id(lock) for

lock in

acquired) >= id(locks[0]):
        raise RuntimeError

('Lock Order Violation')



# Acquire all of the locks





acquired.extend(locks)



_local.acquired = acquired
    try

:
        for

lock in

locks:



lock.acquire()
        yield
```

```
    finally
:

# Release locks in reverse order of acquisition

        for
lock in
reversed(locks):

lock.release()
        del

acquired[-len(locks):]
```

要使用这个上下文管理器，只用按照正常的方式来分配锁对象，但是当想同一个或多个锁打交道时就使用acquire()函数。例如：

```
import threading
```

```
x_lock = threading.Lock()
y_lock = threading.Lock()

def


thread_1():
    while


True:
        with


acquire(x_lock, y_lock):
            print


('Thread-1')

def


thread_2():
    while


True:
        with


acquire(y_lock, x_lock):
            print


('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果运行这个程序，就会发现程序运行得很好，而且永远不会出现死锁——尽管在每个函数中对锁的获取是以不同的顺序来指定的。

这个例子的关键之处就在于acquire()函数的第一条语句：根据对象的数字编号对锁进行排序。通过对锁进行排序，无论用户按照什么顺序将锁提供给acquire()函数，它们总是会按照统一的顺序来获取。

这个解决方案中用到了线程本地存储（thread-local storage）来解决一个小问题。即，如果有多个acquire()操作嵌套在一起，可以检测可能存在的死锁情况。例如，假设编写了如下的代码：

```
import threading



x_lock = threading.Lock()
y_lock = threading.Lock()

def


thread_1():
    while


True:
```

```
        with


acquire(x_lock):
            with


acquire(y_lock):
                print


('Thread-1')
def


thread_2():
    while


True:
        with


acquire(y_lock):
            with


acquire(x_lock):
                print


('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果运行这个版本的程序，其中一个线程将会因抛出异常而崩溃：

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/local/lib/python3.3/threading.py", line 639, in _boo
    self.run()
  File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
  File "deadlock.py", line 49, in thread_1
    with


acquire(y_lock):
  File "/usr/local/lib/python3.3/contextlib.py", line 48, in __en
    return


next(self.gen)
  File "deadlock.py", line 15, in acquire
    raise RuntimeError


("Lock Order Violation")
RuntimeError: Lock Order Violation
>>>
```

这个崩溃基于这样一个事实，即每个线程都会记住它们已经获取到的锁的顺序。acquire()函数会检查之前获取到的锁的列表，并对锁的顺序做强制性的约束：先获取到的锁的对象ID必须比后获取的锁的ID要小。

12.5.3 讨论

在多线程程序中，死锁是一个老生常谈的问题（也是操作系统教科书上的常见主题）。基本原则就是，只要可以保证线程一次只持有一把锁，那么程序就不会出现死锁。但是，一旦在同一时间获取了多个锁，那么什么事情都有可能发生。

检测死锁和从死锁中恢复是一个极其棘手的问题，而且也很少有优雅的解决方案。比方说，通常用来检测死锁和恢复的方案涉及对看门狗定时器的使用。随着线程的运行，它们会周期性地重置定时器，只要一切都运行正常那么就皆大欢喜。但是，如果程序中出现死锁，看门狗定时器最终就会超时。此时，程序就通过重新启动来完成“恢复”。

而避免死锁采用的则是另一种不同的策略，即，以一种根本就不会让程序进入死锁状态的方式来使用锁。前面介绍的解决方案中，总是严格按照对象ID的升序来获取锁，这个方法可以在数学上证明能够避免死锁状态。我们把证明的过程就留给读者当做练习吧（要点就是，严格以递增的顺序来获取锁就不会出现锁的循环依赖，而这正是出现死锁的必要条件）。

作为最后一个例子，我们将讨论一个经典的线

程死锁问题——即所谓的“哲学家就餐问题”。在这个问题中，有5位哲学家围坐在桌边，桌上有5碗米饭和5支筷子。每位哲学家代表着一个独立的线程，而每支筷子代表一把锁。在这个问题中，哲学家要么坐着思考要么就吃米饭。但是，要吃到米饭，哲学家需要两支筷子。不幸的是，如果所有的哲学家都伸手拿他们左手边的那支筷子，那么他们只能全都坐在那里，手里只拿着一支筷子最终饿死。真是可怕的景象。

采用我们前面的解决方案，下面是哲学家就餐问题的简单实现，可完全避免死锁：

```
import threading



# The philosopher thread



def


philosopher(left, right):
    while


True:
        with


acquire(left,right):
             print
```

```
(threading.currentThread(), 'eating')

# The chopsticks (represented by locks)


NSTICKS = 5
chopsticks = [threading.Lock() for


n in


range(NSTICKS)]

# Create all of the philosophers


for


n in


range(NSTICKS):


t = threading.Thread(target=philosopher,


args=(chopsticks[n],chopsticks[(n+1) % NSTICKS]))


t.start()
```

最后但同样值得注意的是，为了避免死锁，所有的锁都必须使用我们前面给出的acquire()函数来获取。如果某些代码片段中是直接获取锁的，那么这个避免死锁的算法就不能奏效了。

12.6 保存线程专有状态

12.6.1 问题

我们需要保存当前运行线程的专有状态，这个状态对其他线程是不可见的。

12.6.2 解决方案

有时候在多线程程序中，我们需要保存专属于当前运行线程的状态。为了做到这点，可以通过threading.local()来创建一个线程本地存储对象。在这个对象上保存和读取的属性只对当前运行的线程可见，其他线程无法感知。

作为使用线程局部存储的一个有趣实例，考虑一下LazyConnection上下文管理器类，我们在8.3节中首次定义了这个类。下面是稍微修改过的版本，可以安全应用于多线程环境中：

```
from socket import


socket, AF_INET, SOCK_STREAM
import threading
```

```
class LazyConnection

:
    def

__init__(self, address, family=AF_INET, type=SOCK_STREAM):

self.address = address

self.family = AF_INET

self.type = SOCK_STREAM

self.local = threading.local()

    def

__enter__(self):
        if

hasattr(self.local, 'sock'):
            raise RuntimeError

('Already connected')

self.local.sock = socket(self.family, self.type)
```

```
self.local.sock.connect(self.address)
        return

self.local.sock

    def

__exit__(self, exc_ty, exc_val, tb):

self.local.sock.close()
        del

self.local.sock
```

在这份代码中，请仔细观察对self.local属性的使用。它被初始化为threading.local()的实例。之后，其他方法操作的socket都是被保存为self.local.sock的形式。这就足以使得LazyConnection的实例可以安全用于多线程环境中了。示例如下：

```
from functools import

partial
def
```

```
test(conn):
    with

conn as

s:


s.send(b'GET /index.html HTTP/1.0\r\n

')


s.send(b'Host: www.python.org\r\n

')





s.send(b'\r\n

')


resp = b''.join(iter(partial(s.recv, 8192), b''))

    print

('Got {} bytes'.format(len(resp)))

if
```

```
__name__ == '__main__':

conn = LazyConnection(('www.python.org', 80))

t1 = threading.Thread(target=test, args=(conn,))

t2 = threading.Thread(target=test, args=(conn,))

t1.start()

t2.start()

t1.join()

t2.join()
```

这么做能正常工作的原因在于每个线程实际上创建了自己专属的socket连接（以self.local.sock的形式保存）。因此，当不同的线程在socket上执行操作时，它们并不会互相产生影响，因为它们都是在

不同的socket上完成操作的。

12.6.3 讨论

在大部分程序中，创建和操作线程专有状态都不会出现什么问题。但是万一出现问题了，通常是因为多个线程使用了同一个对象，而该对象需要操作某种系统资源，比如说socket或者文件。我们不能让一个单独的socket对象被所有线程共享，因为如果有多个线程同时对socket进行读或写，那么就会出现混乱。线程专有存储通过让这种资源只对一个线程可见，解决了这个问题。

在本节示例中，使用threading.local()使得LazyConnection类支持每个线程一条连接，而不是之前的整个进程就一条连接。这是一个微妙但有趣的区别。

在底层，threading.local()实例为每个线程维护着一个单独的实例字典。所有对实例的常见操作比如获取、设定以及删除都只是作用于每个线程专有的字典上。每个线程使用一个单独的字典，正是这一事实使得不同线程的数据得到隔离。

12.7 创建线程池

12.7.1 问题

我们想创建一个工作者线程池用来处理客户端连接，或者完成其他类型的工作。

12.7.2 解决方案

concurrent.futures库中包含有一个ThreadPoolExecutor类可用来实现这个目的。下面的示例是一个简单的TCP服务器，使用线程池来服务客户端：

```
from socket import


AF_INET, SOCK_STREAM, socket
from concurrent.futures import


ThreadPoolExecutor

def


echo_client(sock, client_addr):



'''
```

```
Handle a client connection

'''

    print

('Got connection from', client_addr)
    while

True:

msg = sock.recv(65536)
        if not

msg:
            break

sock.sendall(msg)
    print

('Client closed connection')
```

```
sock.close()

def

echo_server(addr):

pool = ThreadPoolExecutor(128)

sock = socket(AF_INET, SOCK_STREAM)

sock.bind(addr)

sock.listen(5)
    while

True:

client_sock, client_addr = sock.accept()

pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

如果想手动创建自己的线程池，使用Queue来实现通常是足够简单的。下面的例子对上述代码做了修改，手动实现了线程池：

```
from socket import

socket, AF_INET, SOCK_STREAM
from threading import

Thread
from queue import

Queue

def

echo_client(q):

'''

Handle a client connection

'''
```

```
sock, client_addr = q.get()
    print

('Got connection from', client_addr)
    while

True:


msg = sock.recv(65536)
        if not

msg:
            break




sock.sendall(msg)
    print

('Client closed connection')


sock.close()

    def

echo_server(addr, nworkers):
```

```
# Launch the client workers

q = Queue()
        for

n in

range(nworkers):

t = Thread(target=echo_client, args=(q,))

t.daemon = True

t.start()

# Run the server

sock = socket(AF_INET, SOCK_STREAM)

sock.bind(addr)
```

```
sock.listen(5)
    while

True:

client_sock, client_addr = sock.accept()

q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

应该使用ThreadPoolExecutor而不是手动实现线程池。这么做的优势在于使得任务的提交者能够更容易从调用函数中取得结果。例如，可以像这样编写代码：

```
from concurrent.futures import

ThreadPoolExecutor
import urllib.request

def

fetch_url(url):
```

```
u = urllib.request.urlopen(url)

data = u.read()
    return

data

pool = ThreadPoolExecutor(10)
# Submit work to the pool

a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

# Get the results back

x = a.result()
y = b.result()
```

示例中的结果对象（即，a和b）负责处理所有需要完成的阻塞和同步任务，从工作者线程中取回数据。特别是，a.result()操作会阻塞，直到对应的函数已经由线程池执行完毕并返回了结果为止。

12.7.3 讨论

一般来说，应该避免编写那种允许线程数量无

限增长的程序。比如，看看下面这个服务器实现：

```
from threading import


Thread
from socket import


socket, AF_INET, SOCK_STREAM

def


echo_client(sock, client_addr):


'''





Handle a client connection





'''



    print


('Got connection from', client_addr)
    while
```

```
True:

msg = sock.recv(65536)
        if not

msg:
            break

sock.sendall(msg)
    print

('Client closed connection')

sock.close()

def

echo_server(addr, nworkers):

# Run the server

sock = socket(AF_INET, SOCK_STREAM)
```

```
sock.bind(addr)

sock.listen(5)
    while

True:

client_sock, client_addr = sock.accept()

t = Thread(target=echo_client, args=(client_sock, client_addr))

t.daemon = True

t.start()

echo_server(('',15000))
```

尽管可以工作，但是无法阻止恶意用户对服务器发起拒绝服务攻击，从而导致服务器上创建了大量的线程，耗尽了系统资源而崩溃（因而进一步说明了使用线程的“邪恶”之处）。通过使用预先初始化好的线程池，就可以小心地为所能支持的并发总数设定一个上限值。

我们可能会担心创建大量线程所产生的影响。

但是，在现代的系统上创建拥有几千个线程的线程池是不会有什么问题的。此外，让一千个线程等待工作并不会对其他部分的代码产生性能上的影响（休眠的线程什么也不做）。当然了，如果所有这些线程在同一时间被唤醒开始使用CPU的话那就是另一回事了 [2] ——尤其在有全局解释器锁（GIL）的情况下更是如此。一般来说，线程池只适用于处理I/O密集型的任务。

创建大型的线程池需要考虑的一个方面就是对内存的使用。比如说，在OS X上如果创建了两千个线程，系统显示Python进程占用了超过9 GB的虚拟内存。但是，这实际上是有一些误导的。当创建一个线程时，操作系统会占用一段虚拟内存来保存线程的执行栈（通常有8 MB）。这段内存只有一小部分会实际映射到物理内存上。因此，如果看的更仔细一些，就会发现Python进程占用的物理内存远比虚拟内存要小（例如，创建两千个线程只使用了70 MB物理内存，不是9 GB）。如果需要考虑虚拟内存的大小，可以使用threading.stack_size()函数来将栈的大小调低。例如：

```
import threading

threading.stack_size(65536)
```

如果增加这个调用，然后重复试验创建两千个线程，就会发现Python进程现在只使用了大约210 MB虚拟内存，但使用的物理内存总量保持不变。注意，线程栈的大小必须至少有32768字节，通常会限制该值为系统内存页面大小（40968192等）的整数倍。

12.8 实现简单的并行编程

12.8.1 问题

我们有一个执行了大量CPU密集型工作的程序，现在想让它利用多个CPU的优势运行得更快些。

12.8.2 解决方案

concurrent.futures库中提供了一个ProcessPoolExecutor类，可用来在单独运行的Python解释器实例中执行计算密集型的函数。但是为了使用这个功能，首先得有一些计算密集型的任务才行。让我们以一个简单但有实际意义的例子来说明。

假设有一个目录，里面全是gzip压缩格式的Apache Web服务器的日志文件：

```
logs/

20120701.log.gz
```

```
20120702.log.gz

20120703.log.gz

20120704.log.gz

20120705.log.gz

20120706.log.gz
    ...
```

进一步假设每个日志文件中包含有如下这样的文本行：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt .
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 2
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.x
...
```

下面是一个简单的脚本，它读取数据并标识出所有访问过robots.txt文件的主机：

```
# findrobots.py
```

```
import gzip

import io

import glob

def
find_robots(filename):
'''

Find all of the hosts that access robots.txt in a single log file

'''
```

```
robots = set()
    with

gzip.open(filename) as

f:
        for

line in

io.TextIOWrapper(f,encoding='ascii'):


fields = line.split()
            if

fields[6] == '/robots.txt':


robots.add(fields[0])
    return

robots

def

find_all_robots(logdir):


'''
```

```
Find all hosts across and entire sequence of files

'''

files = glob.glob(logdir+'/*.log.gz')

all_robots = set()
    for

robots in

map(find_robots, files):

all_robots.update(robots)
    return

all_robots

if

__name__ == '__main__':

robots = find_all_robots('logs')
    for
```

```
ipaddr in

robots:
        print

(ipaddr)
```

上面的程序以常用的map-reduce风格[3]来编写。函数find_robots()被映射到一系列的文件名上，将所有得到的结果合并成一个单独的结果（即find_all_robots()函数中设置的all_robots）。

现在假设想修改这个程序以利用多个CPU核心。这是很容易实现的——只需把map()替换成一个类似的操作，并让它在concurrent.futures库中的进程池中执行即可。下面是稍微修改过的代码：

```
# findrobots.py

import gzip

import io
```

```
import glob

from concurrent import

futures
def

find_robots(filename):

'''

Find all of the hosts that access robots.txt in a single log fil

'''

robots = set()
    with

gzip.open(filename) as
```

```
f:
        for

line in

io.TextIOWrapper(f,encoding='ascii'):


fields = line.split()
            if

fields[6] == '/robots.txt':


robots.add(fields[0])
    return

robots

def

find_all_robots(logdir):


'''






Find all hosts across and entire sequence of files
```

```
'''




files = glob.glob(logdir+'/*.log.gz')


all_robots = set()
    with


futures.ProcessPoolExecutor() as


pool:
        for


robots in


pool.map(find_robots, files):



all_robots.update(robots)
    return


all_robots

if


__name__ == '__main__':



robots = find_all_robots('logs')
    for
```

```
ipaddr in

robots:
        print

(ipaddr)
```

经过这次修改，现在这个脚本在我们的四核机器上运行时要比之前的版本快3.5倍，得到的结果完全相同。实际的性能会根据机器上的CPU个数而有所不同。

12.8.3 讨论

ProcessPoolExecutor的典型用法是这样的：

```
from concurrent.futures import

ProcessPoolExecutor
with

ProcessPoolExecutor() as

pool:
```

```
...

do work in

parallel using pool
    ...
```

在底层，ProcessPoolExecutor创建了N个独立运行的Python解释器，这里的N就是在系统上检测到的可用的CPU个数。可以修改创建的Python进程个数，只要给ProcessPoolExecutor(N)提供一个可选的参数即可。进程池会一直运行，直到with语句块中的最后一条语句执行完毕为止，此时进程池就会关闭。但是，程序会一直等待所有已经提交的任务都处理完毕为止。

提交到进程池中的任务必须定义成函数的形式。有两种方法可以提交任务。如果想并行处理一个列表推导式或者map()操作，可以使用pool.map()：

```
# A function that performs a lot of work
```

```
def

work(x):


...
    return

result
# Nonparallel code


results = map(work, data)
# Parallel implementation


with

ProcessPoolExecutor() as

pool:


results = pool.map(work, data)
```

还有一种方式就是可以通过pool.submit()方法来手动提交一个单独的任务：

```
# Some function

def

work(x):

...
    return

result
with

ProcessPoolExecutor() as

pool:

...

# Example of submitting work to the pool

future_result = pool.submit(work, arg)

# Obtaining the result (blocks until done)
```

```
r = future_result.result()
    ...
```

如果手动提交任务，得到的结果就是一个Future实例。要获取到实际的结果还需要调用它的result()方法。这么做会阻塞进程，直到完成了计算并将结果返回给进程池为止。

与其让进程阻塞，也可以提供一个回调函数，让它在任务完成时得到触发执行。示例如下：

```
def

when_done(r):
    print

('Got:', r.result())

with

ProcessPoolExecutor() as

pool:
```

```
future_result = pool.submit(work, arg)

future_result.add_done_callback(when_done)
```

用户提供的回调函数需要接受一个Future实例，必须用它才能获取到实际的结果（即，调用它的result()方法）。

尽管进程池使用起来很简单，但是在设计规模更大的程序时有几个重要的因素需要考虑。我们在这里说明一下，各因素间不分先后顺序。

- 这种并行化处理的技术只适用于可以将问题分解成各个独立部分的情况。

- 任务必须定义成普通的函数来提交。实例方法、闭包或者其他类型的可调用对象都是不支持并行处理的。

- 函数的参数和返回值必须可兼容于pickle编码。任务的执行是在单独的解释器进程中完成的，这中间需要用到进程间通信。因此，在不同的解释器间交换数据必须要进行序列化处理。

- 提交的工作函数不应该维护持久的状态或者带有副作用。除了简单的日志功能外，一旦子进程启动，将无法控制它的行为。因此，为了让思路保持清晰，最好让每件事情都保持简单，让任务在不会修改执行环境的纯函数（pure-function）中执行。

- 进程池是通过调用UNIX上的fork()系统调用来创建的。这么做会克隆出一个Python解释器，在fork()时会包含所有的程序状态。在Windows上，这么做会加载一个独立的解释器拷贝，但并不包含状态。克隆出来的进程在首次调用pool.map()或者pool.submit()方法之前不会实际运行。

- 当将进程池和多线程技术结合在一起时需要格外小心。特别是，很可能我们应该在创建任何线程之前优先创建并加载进程池（例如，当程序启动时在主线程中创建进程池）。

12.9　如何规避GIL带来的限制

12.9.1　问题

我们已经听说过全局解释器锁（GIL），担心它会影响到多线程程序的性能。

12.9.2　解决方案

尽管Python完全支持多线程编程，但是在解释器的C语言实现中，有一部分并不是线程安全的，因此不能完全支持并发执行。事实上，解释器被一个称之为全局解释器锁（GIL）的东西保护着，在任意时刻只允许一个Python线程投入执行。GIL带来的最明显的影响就是多线程的Python程序无法充分利用多个CPU核心带来的优势（即，一个采用多线程技术的计算密集型应用只能在一个CPU上运行）。

在讨论规避GIL的常用方案之前，需要重点强调的是，GIL只会对CPU密集型的程序产生影响（即，主要完成计算任务的程序）。如果我们的程序主要是在做I/O操作，比如处理网络连接，那么选择多线程技术常常是一个明智的选择。因为它们

大部分时间都花在等待对方发起连接上了。实际上可以创建数以千计的Python线程，一点问题都没有。在现代的操作系统上运行这么多线程是不会有问题的，因此这不是应该担心的地方。

对于CPU密集型的程序，我们需要对问题的本质做些研究。例如，仔细选择底层用到的算法，这可能会比尝试将一个没有优化过的算法用多线程来并行处理所带来的性能提升要高得多。同样地，由于Python是解释型语言，往往只要简单地将性能关键的代码转移到用C语言扩展的模块中就可能得到极大的速度提升。类似NumPy这样的扩展模块对于加速涉及数组数据的特定计算也是非常高效的。最后但同样重要的是，还可以尝试其他的解释器实现，比如说使用了JIT编译优化技术的PyPy（尽管在写作本书时PyPy还没有支持Python 3）。

同样值得指出的是，使用多线程技术并不只是为了获得性能的提升。一个CPU密集型的程序可能会用多线程来管理图形用户界面、网络连接或者其他类型的服务。在这种情况下GIL实际上会带来更多的问题。因为如果某部分代码持有GIL锁的时间过长，那就会导致其他非CPU密集型的线程都阻塞住，这实在令人讨厌。实际上，一个写的很糟糕的C语言扩展模块会让这个问题变得更加严重，尽管代码中用C实现的部分会比之前要运行得更快[4]。

说了这么多，要规避GIL的限制主要有两种常用的策略。第一，如果完全使用Python来编程，可以使用multiprocessing模块来创建进程池，把它当做协处理器来使用。例如，假设线程代码是这样的：

```
# Performs a large calculation (CPU bound)

def

some_work(args):
    ...
    return

result

# A thread that calls the above function

def

some_thread():
    while

True:
        ...
        r = some_work(args)
        ...
```

下面的示例告诉我们如何将代码修改为使用进程池的方式：

```
# Processing pool (see below for initiazation)



pool = None

# Performs a large calculation (CPU bound)



def


some_work(args):
    ...
    return


result

# A thread that calls the above function



def


some_thread():
    while


True:
        ...
        r = pool.apply(some_work, (args))
        ...

# Initiaze the pool
```

```
if

__name__ == '__main__':
    import multiprocessing

    pool = multiprocessing.Pool()

```

这个使用进程池的例子通过一个巧妙的办法避开了GIL的限制。每当有线程要执行CPU密集型的任务时，它就把任务提交到池中，然后进程池将任务转交给运行在另一个进程中的Python解释器。当线程等待结果的时候就会释放GIL。此外，由于计算是在另一个单独的解释器中进行的，这就不再受到GIL的限制了。在多核系统上，将会发现采用这种技术能轻易地利用到所有的CPU核心。

第二种方式是把重点放在C语言扩展编程上。主要思想就是将计算密集型的任务转移到C语言中，使其独立于Python，在C代码中释放GIL。这是通过在C代码中插入特殊的宏来实现的：

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
```

```
    // Threaded C code



    ...
    Py_END_ALLOW_THREADS
    ...
}
```

如果使用其他的工具来访问C代码，比如ctypes库或者Cython，那么可能不需要做任何处理。比方说，ctypes默认会在调用C代码时自动释放GIL。

12.9.3　讨论

有许多程序员每当面对多线程程序性能方面的问题时，总是抱怨GIL是所有问题的根源。但是，这么做只是一种短视和幼稚的行为。举个现实中的例子吧，在多线程网络程序中出现神秘的“僵死”现象很可能是由于和GIL风马牛不相及的原因所造成的（例如，DNS查询失败）。底线就是你需要认真研究自己的代码，判断GIL是否才是问题的原因。再次申明，CPU密集型的处理才需要考虑GIL，I/O密集型的处理则不必。

如果打算使用进程池来规避GIL，这需要涉及同另一个Python解释器之间进行数据序列化和通信

的处理。为了让这种方法奏效，待执行的操作需要包含在以def语句定义的Python函数中（即，在这里lambda、闭包、可调用实例都是不可以的），而且函数参数和返回值必须兼容于pickle编码。此外，要完成的工作规模必须足够大，这样可以弥补额外产生的通信开销。

将多线程和进程池混在一起使用绝对是个让人头痛的好办法。如果打算将这些功能结合在一起使用，通常最好在创建任何线程之前将进程池作为单例（singleton）在程序启动的时候创建。之后，线程就可以使用相同的进程池来处理所有那些计算密集型的工作。

对于C语言扩展模块，最重要的功能就是保持与Python解释器进程的隔离。也就是说，如果打算将任务从Python中转移到C来处理，需要确保C代码可以独立于Python执行。这意味着不使用Python的数据结构，也不调用Python的C语言API。另一个需要考虑的就是要确保编写C语言扩展模块能够完成足够多的任务，这样才值得这么做。也就是说，这个扩展模块最好可以执行几百万次的计算，而不仅仅只是完成几个小规模的计算。

不用说，这些规避GIL限制的解决方案并非对所有问题都适用。例如，某些特定类型的应用如果

分解到多个进程中处理，或者是将部分代码用C来实现，效果都不会很好。对于这些类型的应用，需要找到自己的解决方案（例如，多个进程访问共享的内存区域、让多个解释器运行在同一个进程中，等等）。作为备选方案，我们还可以选择其他的解释器实现，比如PyPy。

有关在C语言扩展中释放GIL的附加内容，可参见15.7节和15.10节。

12.10　定义一个Actor任务

12.10.1　问题

我们想要定义行为上类似于actor的任务，即采用所谓的actor模式来编程。

12.10.2　解决方案

actor模式是最古老也是最简单的用来解决并发和分布式计算问题的方法之一。实际上，actor模式所暗含的简单性正是它的吸引力所在。总的来说，actor就是一个并发执行的任务，它只是简单地对发送给它的消息进行处理。作为对这些消息的响应，actor会决定是否要对其他的actor发送进一步的消息。actor任务之间的通信是单向且异步的。因此，消息的发送者并不知道消息何时才会实际传递，当消息已经处理完毕时也不会接收到响应或者确认。

把线程和队列结合起来使用很容易定义出actor。示例如下：

```
from queue import
```

```
Queue
from threading import


Thread, Event

# Sentinel used for shutdown



class ActorExit


(Exception


):
    pass




class Actor


:
    def


__init__(self):
        self._mailbox = Queue()

    def


send(self, msg):
        '''



        Send a message to the actor



        '''
```

```
        self._mailbox.put(msg)

    def


recv(self):
        '''



        Receive an incoming message



        '''




        msg = self._mailbox.get()
        if


msg is


ActorExit:
            raise


ActorExit()
        return


msg

    def


close(self):
        '''
```

```
        Close the actor, thus shutting it down

        '''

        self.send(ActorExit)

    def

start(self):
        '''

        Start concurrent execution

        '''

        self._terminated = Event()
        t = Thread(target=self._bootstrap)
        t.daemon = True
        t.start()

    def

_bootstrap(self):
        try

:
            self.run()
        except

ActorExit:
```

```
            pass

        finally

:
            self._terminated.set()

    def

join(self):
        self._terminated.wait()

    def

run(self):
        '''

        Run method to be implemented by the user

        '''

        while

True:
            msg = self.recv()

# Sample ActorTask

class PrintActor
```

```
(Actor):
    def


run(self):
        while


True:
            msg = self.recv()
            print


('Got:', msg)

# Sample use




p = PrintActor()
p.start()
p.send('Hello')
p.send('World')
p.close()
p.join()
```

在这个示例中，我们使用actor实例的send()方法来发送消息。在底层，这会将消息放入到队列上，内部运行的线程会从队列中取出收到的消息处理。close()方法通过在队列中放置一个特殊的终止值（ActorExit）来关闭actor。用户可以通过继承Actor类来定义新的actor，并重新定义run()方法来实现自定义的处理。用户自定义的代码可通过ActorExit异常来捕获终止请求，如果合适的话可以处理这个异常（ActorExit异常是在recv()方法中抛

出并传播的）。

如果将并发和异步消息传递的需求去掉，那么完全可以用生成器来定义一个最简化的actor对象。示例如下：

```
def

print_actor():
    while

True:
        try

:
            msg = yield

# Get a message


            print

('Got:', msg)
        except GeneratorExit

:
            print

('Actor terminating')
# Sample use
```

```
p = print_actor()
next(p)         # Advance to the yield (ready to receive)



p.send('Hello')
p.send('World')
p.close()
```

12.10.3 讨论

actor模式之所以吸引人，部分原因是由于它的简单性。在实践中只有一个核心的操作，那就是send()。此外，在基于actor模式的系统中，“消息”的概念可以扩展到许多不同的方向。比方说，可以以元组的形式传递带标签的消息，让actor执行不同的操作：

```
class TaggedActor


(Actor):
    def


run(self):
        while


True:
            tag, *payload = self.recv()
            getattr(self,'do_'+tag)(*payload)
```

```
    # Methods correponding to different message tags


    def


do_A(self, x):
        print


('Running A', x)

    def


do_B(self, x, y):
        print


('Running B', x, y)

# Example



a = TaggedActor()
a.start()
a.send(('A', 1))         # Invokes do_A(1)



a.send(('B', 2, 3))        # Invokes do_B(2,3)
```

再看另一个例子。这里有一个actor的变种，允许在工作者线程中执行任意的函数，并通过特殊的

Result对象将结果回传：

```
from threading import


Event
class Result



:
    def



__init__(self):
        self._evt = Event()
        self._result = None

    def



set_result(self, value):
        self._result = value
        self._evt.set()

    def



result(self):
        self._evt.wait()
        return



self._result

class Worker



(Actor):
    def



submit(self, func, *args, **kwargs):
```

```
        r = Result()
        self.send((func, args, kwargs, r))
        return


r

    def


run(self):
        while


True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

# Example use



worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
print


(r.result())
```

最后但同样重要的是，给任务“发送”一条消息这个概念是可以扩展到涉及多进程甚至是大型的分布式系统中的。例如，可以把actor对象的send()方法实现为在socket连接上传输数据，或者通过某种消息传递的基础架构（比如AMQP、ZMQ等）来完成传递。

12.11　实现发布者/订阅者消息模式

12.11.1　问题

我们要解决一个基于多线程间通信的问题，希望实现发布者/订阅者的消息模式。

12.11.2　解决方案

要实现发布者/订阅者消息模式，一般来说需要引入一个单独的“交换”或者“网关”这样的对象，作为所有消息的中介。也就是说，不是直接将消息从一个任务发往另一个任务，而是将消息发往交换中介，由中介将消息转发给一个或多个相关联的任务。下面给出了一个非常简单的消息交换的实现：

```
from collections import

defaultdict
class Exchange

:
    def
```

```
__init__(self):
        self._subscribers = set()

    def


attach(self, task):
        self._subscribers.add(task)

    def


detach(self, task):
        self._subscribers.remove(task)

    def


send(self, msg):
        for


subscriber in


self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges



_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name



def


get_exchange(name):
    return
```

```
_exchanges[name]
```

交换中介其实就是一个对象，它保存了活跃的订阅者集合，并提供关联、取消关联以及发送消息的方法。每个交换中介都由一个名称来标识，get_exchange()函数简单地返回同给定的名称相关联的那个Exchange对象。

下面是一个简单的例子，展示了如何使用交换中介：

```
# Example of a task. Any object with a send() method



class Task


:
    ...
    def


send(self, msg):
        ...

task_a = Task()
task_b = Task()

# Example of getting an exchange
```

```
exc = get_exchange('name')

# Examples of subscribing tasks to it



exc.attach(task_a)
exc.attach(task_b)

# Example of sending messages



exc.send('msg1')
exc.send('msg2')

# Example of unsubscribing



exc.detach(task_a)
exc.detach(task_b)
```

尽管关于这个主题还有许多不同的变种，但总体思路都是一样的。消息会先传递到一个中介，再由中介将消息传递给相关联的订阅者。

12.11.3 讨论

任务或者线程之间互相发送消息（通常以队列来实现），这个概念很流行也很容易实现。但是，如果使用订阅者/发布者模型来取代传统的做法，带来的好处常常被人们忽视。

首先，使用交换中介可以简化很多设定线程通信的工作。与其通过多个模块将线程连接在一起，现在只需要关心将线程连接到一个已知的交换中介上就行了。从某种意义上说这和logging库的工作方式很相似。在实践中，这么做可以使解耦程序中的多个任务变得更加容易。

其次，交换中介具有将消息广播发送给多个订阅者的能力，这打开了新通信模式的大门。比如说，我们可以实现带有冗余任务、广播或者扇出（fan-out）的系统。也可以构建调试以及诊断工具，将它们作为普通的订阅者关联到交换中介上。下面是一个简单的诊断类，可以显示发送的消息：

```
class DisplayMessages


:
    def


__init__(self):
        self.count = 0
    def


send(self, msg):
        self.count += 1
        print


('msg[{}]: {!r}'.format(self.count, msg))

exc = get_exchange('name')
```

```
d = DisplayMessages()
exc.attach(d)
```

最后但同样重要的是，这种实现方式有一个显著的方面就是它能和各种类似于任务的对象一起工作。比如，消息的接收者可以是actor（12.10节中描述了actor任务）、协程、网络连接，甚至只要实现了合适的send()方法的对象都可以。

关于交换中介，一个可能存在的问题就是如何以适当的方式对订阅者进行关联和取消关联处理。为了能正确管理资源，每个已经关联上的订阅者最终都必须取消关联。这就导致出现类似于下面示例的编程模型：

```
exc = get_exchange('name')
exc.attach(some_task)
try


:
    ...
finally


:
    exc.detach(some_task)
```

从某种意义上说，这和使用文件、锁以及类似的资源对象很相似。经验告诉我们，程序员常常会忘记最后的detach()。为了简化这个步骤，或许会考虑使用上下文管理协议。例如，为交换中介添加一个subscribe()方法：

```
from contextlib import

contextmanager
from collections import

defaultdict

class Exchange

:
    def

__init__(self):
        self._subscribers = set()

    def

attach(self, task):
        self._subscribers.add(task)

    def

detach(self, task):
        self._subscribers.remove(task)

@contextmanager
def
```

```
subscribe(self, *tasks):
    for

task in

tasks:
        self.attach(task)
    try

:
        yield



    finally

:
        for

task in

tasks:
            self.detach(task)

    def

send(self, msg):
        for

subscriber in

self._subscribers:
            subscriber.send(msg)
```

```
# Dictionary of all created exchanges


_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name


def


get_exchange(name):
    return


_exchanges[name]

# Example of using the subscribe() method


exc = get_exchange('name')
with


exc.subscribe(task_a, task_b):
    ...
    exc.send('msg1')
    exc.send('msg2')
    ...

# task_a and task_b detached here
```

最后应该要提到的是，对于交换中介这个思想其实还有许多种可能的扩展。例如，交换中介可以

实现一整个消息通道的集合，或者对交换中介的名称施加模式匹配的规则。交换中介也可以扩展到分布式计算的应用中去（例如，将消息在不同机器上的任务之间进行路由）。

12.12 使用生成器作为线程的替代方案

12.12.1 问题

我们想用生成器（协程）作为系统线程的替代方案来实现并发。协程有时也称为用户级线程或绿色线程。

12.12.2 解决方案

要利用生成器来实现自己的并发机制，首选需要对生成器函数和yield语句的基本原理有所了解。特别是关于yield的基本行为，即，使得生成器暂停执行。由于可以暂停执行，因此可以编写一个调度器将生成器函数当做一种“任务”来对待，并通过使用某种形式的任务切换来交替执行这些任务。

为了说明这个思想，考虑下面两个生成器函数：

```
# Two simple generator functions
```

```
def


countdown(n):
    while


n > 0:
        print


('T-minus', n)
        yield



        n -= 1
    print


('Blastoff!')

def


countup(n):
    x = 0
    while


x < n:
        print


('Counting up', x)
        yield



        x += 1
```

这些函数可能看起来有些滑稽，因为它们都使用了单独的yield语句。但是请考虑下面的代码，我们给出了一个简单的任务调度器实现：

```
from collections import

deque
class TaskScheduler

:
    def

__init__(self):
        self._task_queue = deque()
    def

new_task(self, task):
        '''

        Admit a newly started task to the scheduler

        '''

        self._task_queue.append(task)
    def

run(self):
```

```
        '''
        Run until there are no more tasks
        '''
        while
self._task_queue:
            task = self._task_queue.popleft()
            try
:
                # Run until the next yield statement
                next(task)
                self._task_queue.append(task)
            except StopIteration
:
                # Generator is no longer executing
                pass

# Example use
sched = TaskScheduler()
```

```
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

在这份代码中，TaskScheduler类以循环的方式运行了一系列的生成器函数——每个都运行到yield语句就暂停。例如，上面程序的输出如下：

```
T-minus 10
T-minus 5
Counting up 0
T-minus 9
T-minus 4
Counting up 1
T-minus 8
T-minus 3
Counting up 2
T-minus 7
T-minus 2
...
```

此时如果愿意的话，已经基本上实现了一个微型“操作系统”的核心。生成器函数就是任务，而yield语句就是通知任务需要暂停挂起的信号。调度器只是简单地轮流执行所有的任务，直到没有一个任务还能执行为止。

在实践中，我们很可能不会用生成器来实现像示例这么简单的并发处理。相反，当实现actor或者

网络服务器时，可能会用生成器来取代线程。

下面的代码用生成器来实现actor，完全没有用到线程：

```
from collections import

deque
class ActorScheduler

:
    def

__init__(self):
        self._actors = { }         # Mapping of names to actors

        self._msg_queue = deque()     # Message queue


    def

new_actor(self, name, actor):
        '''

        Admit a newly started actor to the scheduler and give it

        '''
```

```
        self._msg_queue.append((actor,None))
        self._actors[name] = actor

    def

send(self, name, msg):
        '''



        Send a message to a named actor



        '''



        actor = self._actors.get(name)
        if

actor:
            self._msg_queue.append((actor,msg))

    def

run(self):
        '''



        Run as long as there are pending messages.



        '''
```

```
        while

self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try

:
                 actor.send(msg)
            except StopIteration

:
                 pass



# Example use



if

__name__ == '__main__':
    def

printer():
        while

True:
            msg = yield


            print

('Got:', msg)

    def
```

```
counter(sched):
        while

True:
            # Receive the current count

            n = yield

            if

n == 0:
                break

            # Send to the printer task

            sched.send('printer', n)
            # Send the next count to the counter task (recursive)

            sched.send('counter', n-1)

sched = ActorScheduler()
# Create the initial actors

sched.new_actor('printer', printer())
sched.new_actor('counter', counter(sched))

# Send an initial message to the counter to initiate
```

```
sched.send('counter', 10000)
sched.run()
```

这段代码的执行流程可能需要研究一番，但是关键点就在于挂起的消息组成的队列上。基本上，只要有消息需要传递，调度器就会运行。这里有一个值得注意的特性，counter生成器发送消息给自己并进入一个递归循环，但却并不会受到Python的递归限制。

下面有一个高级的示例，展示了如何使用生成器来实现一个并发型的网络应用：

```
from collections import


deque
from select import


select

# This class represents a generic yield event in the scheduler



class YieldEvent


:
    def
```

```
handle_yield(self, sched, task):
        pass



    def


handle_resume(self, sched, task):
        pass




# Task Scheduler


class Scheduler


:
    def


__init__(self):
        self._numtasks = 0         # Total num of tasks



        self._ready = deque()         # Tasks ready to run



        self._read_waiting = {}         # Tasks waiting to read



        self._write_waiting = {}     # Tasks waiting to write




# Poll for I/O events and restart waiting tasks
```

```
def _iopoll(self):
    rset,wset,eset = select(self._read_waiting,
                            self._write_waiting,[])
    for r in rset:
        evt, task = self._read_waiting.pop(r)
        evt.handle_resume(self, task)
    for w in wset:
        evt, task = self._write_waiting.pop(w)
        evt.handle_resume(self, task)

def new(self,task):
    '''
    Add a newly started task to the scheduler
    '''
    self._ready.append((task, None))
```

```
    self._numtasks += 1

def

add_ready(self, task, msg=None):
    '''

    Append an already started task to the ready queue.

    msg is what to send into the task when it resumes.

    '''

    self._ready.append((task, msg))

# Add a task to the reading set

def

_read_wait(self, fileno, evt, task):
    self._read_waiting[fileno] = (evt, task)

# Add a task to the write set

def

_write_wait(self, fileno, evt, task):
    self._write_waiting[fileno] = (evt, task)

def
```

```
run(self):
    '''



    Run the task scheduler until there are no tasks



    '''




    while


self._numtasks:
        if not


self._ready:
             self._iopoll()
        task, msg = self._ready.popleft()
        try



:
            # Run the coroutine to the next yield




            r = task.send(msg)
            if



isinstance(r, YieldEvent):
                r.handle_yield(self, task)
            else



:
                raise RuntimeError
```

```
('unrecognized yield event')
        except StopIteration


:
            self._numtasks -= 1

# Example implementation of coroutine-based socket I/O



class ReadSocket


(YieldEvent):
    def


__init__(self, sock, nbytes):
        self.sock = sock
        self.nbytes = nbytes
    def


handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def


handle_resume(self, sched, task):
        data = self.sock.recv(self.nbytes)
        sched.add_ready(task, data)

class WriteSocket


(YieldEvent):
    def


__init__(self, sock, data):
        self.sock = sock
        self.data = data
    def
```

```
handle_yield(self, sched, task):
        sched._write_wait(self.sock.fileno(), self, task)
    def


handle_resume(self, sched, task):
        nsent = self.sock.send(self.data)
        sched.add_ready(task, nsent)

class AcceptSocket


(YieldEvent):
    def


__init__(self, sock):
        self.sock = sock
    def


handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def


handle_resume(self, sched, task):
        r = self.sock.accept()
        sched.add_ready(task, r)

# Wrapper around a socket object for use with yield



class Socket


(object):
    def


__init__(self, sock):
        self._sock = sock
```

```
    def

recv(self, maxbytes):
        return

ReadSocket(self._sock, maxbytes)
    def

send(self, data):
        return

WriteSocket(self._sock, data)
    def

accept(self):
        return

AcceptSocket(self._sock)
    def

__getattr__(self, name):
        return

getattr(self._sock, name)

if

__name__ == '__main__':
    from socket import

socket, AF_INET, SOCK_STREAM
    import time
```

```
    # Example of a function involving generators. This should


    # be called using line = yield from readline(sock)


    def


readline(sock):
        chars = []
        while


True:
            c = yield


sock.recv(1)
            if not


c:
                break



            chars.append(c)
            if


c == b'\n


':
                break



        return
```

```
b''.join(chars)

# Echo server using generators



class EchoServer


:
    def


__init__(self,addr,sched):
        self.sched = sched
        sched.new(self.server_loop(addr))

    def


server_loop(self,addr):
        s = Socket(socket(AF_INET,SOCK_STREAM))
        s.bind(addr)
        s.listen(5)
        while


True:
            c,a = yield


s.accept()
            print


('Got connection from ', a)
            self.sched.new(self.client_handler(Socket(c)))

    def


client_handler(self,client):
        while
```

```
True:
            line = yield from readline


(client


)
            if not


line:
                break



            line = b'GOT:' + line
            while


line:
                nsent = yield


client.send(line)
                line = line[nsent:]
        client.close()
        print


('Client closed')

sched = Scheduler()
EchoServer(('',16000),sched)
sched.run()
```

这段代码显然需要花上一段时间来仔细研究。然而，这基本上实现了一个小型的操作系统。有一个队列（以deque实现）用来保存处于就绪态的任

务，还有等候区（以字典实现）用来保存因为等待I/O而进入休眠状态的任务。调度器很大程度上就是把任务在就绪队列和I/O等待区之间来回移动。

12.12.3 讨论

当构建基于生成器的并发框架时，使用一般形式的yield是最为常见的：

```
def


some_generator():
    ...
    result = yield


data
    ...
```

以这种形式使用yield的函数更常被称为“协程”。在调度器内部，yield语句是在循环中按照如下方式来处理的：

```
f = some_generator()

# Initial result. Is None to start since nothing has been compute
```

```
result = None
while


True:
    try



:
        data = f.send(result)
        result = ... do some calculation ...
    except StopIteration



:
        break
```

这里关于result的逻辑似乎有点令人费解。然而，传递给send()的值就是用来定义当yield语句恢复执行后返回的结果。因此，如果有yield语句要返回一个结果作为对之前产生的数据的响应，那么它就会在下一个send()操作中返回。如果某个生成器函数刚刚启动，发送None给它会让它前进到第一个yield语句的位置。

除了可以发送值以外，还可以在生成器上执行close()方法。这么做会导致在yield语句上产生一个无声的GeneratorExit异常，这会终止生成器的执行。如果需要的话，生成器可以捕获这个异常并执行清理操作。也可以使用生成器的throw()方法在yield语句上产生一个任意的异常。任务调度器可能

会使用这个异常给运行中的生成器传达错误信息。

最后那个示例中用到的yield from语句是用来实现协程的，它可以作为子例程（subroutine）或过程（procedure）从其他的生成器中调用。基本上来说，程序的控制流是以透明的方式转移到新的函数中的。与普通的生成器不同的是，采用yield from调用的函数，其返回值可以成为yield from语句的结果。有关yield from的更多信息可以在PEP 380（http://www.python.org/dev/peps/pep-0380 ）中找到。

最后，如果要用生成器来编程，需要重点强调关于生成器的一些主要局限。尤其是，线程所提供的优势在生成器中都不复存在了。例如，如果执行了任何CPU密集型或者I/O阻塞型的代码，这就会使整个任务调度器挂起，直到完成全部操作为止。为了规避这个限制，唯一的选择就是将这个操作转移到一个可以独立运行的线程或进程中执行。另一个限制在于大多数Python库的实现还不能和基于生成器的线程很好地配合在一起使用。如果选择了这种方法，会发现需要为许多标准库函数编写新的替换版本。有关协程的基础背景和本节中所采用的技术，可以参阅PEP 342（http://www.python.org/dev/peps/pep-0342 ）以及David Beazley在PyCon 2009上做的报告“A

Curious Course On Coroutines and Concurrency”（http://www.dabeaz.com/coroutines ）。

PEP 3156（http://www.python.org/dev/peps/pep-3156 ）中也采用了现代化的方法来处理异步I/O，其中也涉及了协程。在实践中自己编写一个底层的协程调度器是极不可能的。然而，有许多流行的Python程序库都是以协程的思想为基础的，这包括gevent（http://www.gevent.org ）、greenlet（http://pypi.python.org/pypi/greenlet ）、Stackless Python（http://www.stackless.com ）以及其他类似的项目。

12.13　轮询多个线程队列

12.13.1　问题

我们有一组线程队列，想轮询这些队列来获取数据。很大程度上这和轮询一组网络连接来获取数据类似。

12.13.2　解决方案

对于轮询问题，我们常用的解决方案中涉及一个鲜为人知的技巧，即利用隐藏的环回（loopback）网络连接。基本上来说思路是这样的：针对每个想要轮询的队列（或任何对象），创建一对互联的socket。然后对其中一个socket执行写操作，以此表示数据存在。另一个socket就传递给select()或者类似的函数来轮询数据。下面用一些简单的代码来说明这个思路：

```
import queue

import socket
```

```
import os



class PollableQueue


(queue.Queue):
    def


__init__(self):
        super().__init__()
        # Create a pair of connected sockets



        if


os.name == 'posix':
            self._putsocket, self._getsocket = socket.socketpair(
    else


:
        # Compatibility on non-POSIX systems



        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM
        server.bind(('127.0.0.1', 0))
        server.listen(1)
        self._putsocket = socket.socket(socket.AF_INET, socket.SO
        self._putsocket.connect(server.getsockname())
        self._getsocket, _ = server.accept()
        server.close()

def


fileno(self):
    return
```

```
self._getsocket.fileno()

def


put(self, item):
    super().put(item)
    self._putsocket.send(b'x')

def


get(self):
    self._getsocket.recv(1)
    return


super().get()
```

在这份代码中，我们定义了一种新的Queue实例，其底层有一对互联的socket。在UNIX上用socketpair()函数来建立这样的socket对是非常容易的。在Windows上，我们不得不使用示例中展示的方法来伪装socket对（这看起来有些怪异，首先创建一个服务器socket，之后立刻创建客户端socket并连接到服务器上）。之后对get()和put()方法做了些微重构，在这些socket上执行了少量的I/O操作。put()方法在将数据放入队列之后，对其中一个socket写入了一个字节的数据。当要把数据从队列中取出时，get()方法就从另一个socket中把那个单独的字节读出。

fileno()方法使得这个队列可以用类似select()这样的函数来轮询。基本上来说，fileno()方法只是暴露出底层由get()函数所使用的socket的文件描述符。

下面的代码示例定义了一个消费者，用来在多个队列上监视是否有数据到来：

```
import select

import threading


def

consumer(queues):
    '''

    Consumer that reads data on multiple queues simultaneously

    '''

    while

True:
        can_read, _, _ = select.select(queues,[],[])
```

```
        for


r in


can_read:
            item = r.get()
            print


('Got:', item)

q1 = PollableQueue()
q2 = PollableQueue()
q3 = PollableQueue()
t = threading.Thread(target=consumer, args=([q1,q2,q3],))
t.daemon = True
t.start()

# Feed data to the queues




q1.put(1)
q2.put(10)
q3.put('hello')
q2.put(15)
...
```

如果试着运行这段代码，就会发现不管把数据放入到哪个队列中，消费者最后都能接收到所有的数据。

12.13.3　讨论

要对非文件类型的对象比如队列做轮询操作，通常都是看起来简单做起来难。比如说，如果不采用本节展示的socket技术，那唯一的选择就是遍历所有的队列，分别判断每个队列是否为空，而且还得使用定时器（避免CPU利用率达到100%）。就像下面这样：

```
import time



def


consumer(queues):
    while


True:
        for


q in


queues:
            if not


q.empty():
                item = q.get()
                print


('Got:', item)
            # Sleep briefly to avoid 100% CPU
```

```
            time.sleep(0.01)
```

这对于某些特定类型的问题或许是行得通的，但是这么做很笨拙，而且还会引入奇怪的性能方面的问题。例如，如果新的数据添加到了一个队列中，那么至少有10毫秒的时间才能检测到（对于一个现代的处理器来说，10毫秒就好像下辈子那么漫长）。

如果将上面这种轮询方式同其他的轮询对象（比如socket）混在一起使用，那么会遇到更多问题。例如，如果想同时轮询socket和队列，那就不得不使用这样的代码[5]：

```
import select



def


event_loop(sockets, queues):
    while


True:
        # polling with a timeout



```

```
        can_read, _, _ = select.select(sockets, [], [], 0.01)
        for

r in

can_read:
            handle_read(r)
        for

q in

queues:
            if not

q.empty():
                item = q.get()
                print

('Got:', item)
```

本节给出的解决方案解决了很多这样的问题。这就是通过把队列放在和socket同等的地位上来解决的。只要一个单独的select()调用就可以轮询这两种对象的活跃性。不需要使用超时或其他基于时间的技巧来做周期性的检查。此外，如果数据添加到了队列中，消费者几乎能在同一时间得到通知。尽管底层的I/O会带来一点小小的负载（即，在底层的socket对上写入和读出一字节的数据），但由于可以获得更好的响应时间以及简化了代码的编写，

因此这么做通常都是很值得的。

12.14 在UNIX上加载守护进程

12.14.1 问题

我们想编写一个程序，使它能够在UNIX或类UNIX的操作系统上以守护进程的方式运行。

12.14.2 解决方案

创建一个合适的守护进程需要以精确的顺序调用一系列的系统调用，并小心注意其中的细节。下面的代码定义了一个守护进程，附带还有当启动之后可以轻易让它停止运行的能力：

```
#!/usr/bin/env python3

# daemon.py

import os

import sys

```

```
import atexit


import signal



def


daemonize(pidfile, *, stdin='/dev/null',
                          stdout='/dev/null',
                          stderr='/dev/null'):

    if


os.path.exists(pidfile):
        raise RuntimeError


('Already running')

    # First fork (detaches from parent)



    try


:
        if


os.fork() > 0:
            raise SystemExit


(0) # Parent exit



    except OSError as
```

```
e:
        raise RuntimeError


('fork #1 failed.')

    os.chdir('/')
    os.umask(0)
    os.setsid()
    # Second fork (relinquish session leadership)



    try


:
        if


os.fork() > 0:
            raise SystemExit


(0)
    except OSError as


e:
        raise RuntimeError


('fork #2 failed.')

    # Flush I/O buffers



    sys.stdout.flush()
    sys.stderr.flush()

    # Replace file descriptors for stdin, stdout, and stderr
```

```
    with

open(stdin, 'rb', 0) as

f:
        os.dup2(f.fileno(), sys.stdin.fileno())
    with

open(stdout, 'ab', 0) as

f:
        os.dup2(f.fileno(), sys.stdout.fileno())
    with

open(stderr, 'ab', 0) as

f:
        os.dup2(f.fileno(), sys.stderr.fileno())

    # Write the PID file


    with

open(pidfile,'w') as

f:
        print

(os.getpid(),file=f)

    # Arrange to have the PID file removed on exit/signal
```

```
    atexit.register(lambda

: os.remove(pidfile))

    # Signal handler for termination (required)


    def

sigterm_handler(signo, frame):
        raise SystemExit

(1)

    signal.signal(signal.SIGTERM, sigterm_handler)

def

main():
        import time


        sys.stdout.write('Daemon started with pid {}\n

'.format(os.getpid()))
        while

True:
            sys.stdout.write('Daemon Alive! {}\n

'.format(time.ctime()))
            time.sleep(10)

if
```

```
__name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if


len(sys.argv) != 2:
        print


('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr)
        raise SystemExit


(1)

    if


sys.argv[1] == 'start':
        try


:
            daemonize(PIDFILE,
                      stdout='/tmp/daemon.log',
                      stderr='/tmp/dameon.log')
    except RuntimeError as


e:
        print


(e, file=sys.stderr)
        raise SystemExit


(1)

    main()

elif
```

```
sys.argv[1] == 'stop':
    if


os.path.exists(PIDFILE):
        with


open(PIDFILE) as


f:
            os.kill(int(f.read()), signal.SIGTERM)
    else


:
        print


('Not running', file=sys.stderr)
        raise SystemExit


(1)

else


:
    print


('Unknown command {!r}'.format(sys.argv[1]), file=sys.stderr)
    raise SystemExit


(1)
```

要加载这个守护进程，需要使用下面这样的命令：

```
bash % daemon.py start
bash % cat /tmp/daemon.pid
2882
bash % tail -f /tmp/daemon.log
Daemon started with pid 2882
Daemon Alive! Fri Oct 12 13:45:37 2012
Daemon Alive! Fri Oct 12 13:45:47 2012
...
```

守护进程完全在后台运行，因此命令会立刻返回。但是，可以像上面的命令那样检查与守护进程相关的pid文件和日志。要停止这个守护进程，可以这样：

```
bash % daemon.py stop
bash %
```

12.14.3　讨论

本节定义的daemonize()函数可以在程序启动的时候调用，这样可以使程序以守护进程的方式运行。daemonize()的函数签名采用的是keyword-only参数，这样当使用可选参数时能让意图显得更加清晰。这迫使用户必须这样来调用函数：

```
daemonize('daemon.pid',
          stdin='/dev/null,
          stdout='/tmp/daemon.log',
          stderr='/tmp/daemon.log')
```

而不是使用下面这种神秘难懂的调用方式：

```
# Illegal. Must use keyword arguments



daemonize('daemon.pid',
          '/dev/null', '/tmp/daemon.log','/tmp/daemon.log')
```

接下来创建守护进程的步骤就更加晦涩了。基本思想就是，首先，守护进程必须将它自己与父进程分离开来。这就是第一个os.fork()操作完成后立刻终结父进程的目的所在。

当子进程成为孤儿后，就调用os.setsid()创建一个全新的进程会话，并将子进程设为会话的头领。这么做也将子进程设为新的进程组的头领进程，并确保没有任何与之关联的控制终端。如果这听起来显得很神奇，那么这实际上是为了以合适的方式将守护进程从终端中分离开来，并确保像信号这样的东西不会影响到守护进程的运行。

对os.chdir()和os.umask(0)的调用将改变当前的工作目录，并重置文件模式掩码。改变工作目录通常是个好主意，这样守护进程就不再工作于加载它的那个目录之下了。

第二个os.fork()调用是目前为止最为神秘的操作。这一步使得守护进程放弃获取一个新的控制终端的能力，并为自己和外部环境提供了更多的隔离（从根本上说，守护进程会放弃它的会话头领位置，因此不再具有打开控制终端的权限了）。尽管可以忽略这一步，但一般来说还是推荐这么做的。

一旦守护进程已经以合适的方式完成了分离，就执行后续的步骤来重新初始化标准I/O流，使其指向由用户指定的文件。这一部分的处理实际上也需要用到一些技巧。在Python解释器中可以找到多个与标准I/O流相关联的文件对象引用（比如sys.stdout、sys.__stdout__等）。只关闭sys.stdout并为其重新赋值并不能保证可以正常工作，因为没法知道这么做是否可以修改到所有对sys.stdout的引用。相反，我们打开一个单独的文件对象，并通过os.dup2()调用来取代当前由sys.stdout所使用的文件描述符。当这一切发生的时候，sys.stdout原来所用的文件会被关闭，并用新的文件取代它的位置。必须要强调的是，任何已经作用于标准I/O流的文件编码或文本处理将保持原样。

守护进程的一种常见做法就是将它的进程ID写入到一个文件中，以便稍后给其他的程序使用。daemonize()函数最后的部分正是在执行写文件的操作，但同样也做了安排，可以让这个文件在程序终止时被删除。由atexit.register()注册的函数可以在Python解释器退出时得到执行。针对信号SIGTERM所定义的信号处理例程同样需要以优雅得体的方式退出。这个信号处理例程只是发出SystemExit()异常，除此之外什么都不做。这乍看起来是没有必要的，但如果没有这个信号处理例程，终止信号就会杀死解释器进程而不会执行注册到atexit.register()中的清理函数。杀死守护进程的代码示例可以在程序结尾部分处理stop命令的地方找到。

更多关于编写守护进程的信息可以在W.Richard Stevens和Stephen A.Rago合著的*Advanced Programming in the UNIX Environment*，2nd Edition（Addison-Wesley，2005）中找到。尽管此书的重点是用C语言来编程，但所有的内容都很容易适应于Python。因为所有所需的POSIX函数都可以在Python标准库中找到。

---

[1]　因为无法保证发起清除事件请求的线程和再次等待该事件的线程间的执行顺序。——译者注

[2]　即所谓的惊群现象。——译者注

[3]　即函数式编程中的map（映射）和reduce（化简）。——译者注

[4]　要想理解这一段内容，需要对Python解释器的行为有一些了解。对于I/O密集型的线程，每当阻塞等待I/O操作时解释器都会释放GIL。对于从来不执行任何I/O操作的CPU密集型线程，Python解释器会在执行了一定数量的字节码后释放GIL，以便其他线程得到执行的机会。但是C语言扩展模块不同，调用C函数时GIL会被锁定，直到它返回为止。由于C代码的执行是不受解释器控制的，这一期间不会执行任何Python字节码，因此解释器就没法释放GIL了。如果编写C语言扩展时不小心，比方说调用了会阻塞的C函数、执行耗时很长的操作等，那么必须等到C函数返回时才会释放GIL，这时其他的线程就僵死了。这就是为什么作者说如果C扩展模块实现的很糟糕的话，会让问题变得更严重。——译者注

[5]　这种问题的根源就在于没有把队列和socket放在同等的地位上，参见本节开头对问题的描述。——译者注

第**13**章　实用脚本和系统管理

有很多人把Python当做shell脚本的替代，用来实现系统任务的自动化处理，比如操纵文件、配置系统等。本章的主要目标是描述编写脚本时常会遇到的一些任务。比如，解析命令行选项、操纵文件系统中的文件、获取有用的系统配置数据等。本书第5章中也包含了一些与文件和目录相关的信息。

13.1 通过重定向、管道或输入文件来作为脚本的输入

13.1.1 问题

我们希望自己编写的脚本能够接受任意一种对用户来说最为方便的输入机制。这应该包括从命令中产生输出给脚本、把文件重定向到脚本，或者只是在命令行中传递一个或者一列文件名给脚本。

13.1.2 解决方案

Python内置的fileinput模块使得这一切变得非常简单。如果有一个类似下面这样的脚本：

```
#!/usr/bin/env python3

import fileinput

with
fileinput.input() as
```

```
f_input:
     for

line in

f_input:
          print

(line, end='')
```

那么已经可以让脚本按照上述所有的方式来接收输入了。如果将这个脚本保存为*filein.py* 并使其成为可执行的，那么就能够完成下列所有的操作并得到期望的输出：

```
$ ls | ./filein.py            # Prints a directory listing to st

$ ./filein.py /etc/passwd         # Reads /etc/passwd to stdout.

$ ./filein.py < /etc/passwd     # Reads /etc/passwd to stdout.
```

13.1.3 讨论

函数fileinput.input()创建并返回一个FileInput类的实例。除了包含有一些方便实用的帮助函数外，该实例还可以当做上下文管理器来用。因此，把所有这些结合在一起，如果我们编写一个脚本期望它能立刻从多个文件中打印输出，我们可以在输出中包含文件名和行号信息，就像下面这样：

```
>>> import fileinput



>>> with


fileinput.input('/etc/passwd') as


f:
>>>      for


line in


f:
...          print


(f.filename(), f.lineno(), line, end='')
...
/etc/passwd 1 ##



/etc/passwd 2 # User Database

```

```
/etc/passwd 3 #

<other output omitted>
```

把它当做上下文管理器来使用可确保文件不再使用时会被关闭，此外我们这里还利用了FileInput实例的帮助函数来获取一些额外的信息。

13.2　终止程序并显示错误信息

13.2.1　问题

我们想让自己的程序在终止时向标准错误输出打印一条消息并返回一个非零的状态码。

13.2.2　解决方案

要让程序以这种方式终止，可以发出一个SystemExit异常，但是要提供错误信息作为参数。示例如下：

```
raise SystemExit('It failed!')
```

这会导致提供的信息被打印到sys.stderr上，且程序退出时的状态码为1。

13.2.3　讨论

本节的内容十分短小，但是解决了编写脚本时的一个常见问题。即，要终止一个程序，以前可能会倾向于编写这样的代码：

```
import sys

sys.stderr.write('It failed!\n

')
raise SystemExit

(1)
```

现在，不需要再同这些import或者sys.stderr搅和在一起了，只需要给SystemExit()提供一条错误信息即可。

13.3 解析命令行选项

13.3.1 问题

我们想编写一个程序用来解析在命令行中提供的各种选项（选项保存在sys.argv中）。

13.3.2 解决方案

可以用argparse模块来解析命令行选项。我们用一个简单的例子来帮助说明这里的核心特性：

```
# search.py

'''

Hypothetical command-line tool for searching a collection of

files for one or more text patterns.

'''

```

```
import argparse



parser = argparse.ArgumentParser(description='Search some files'

parser.add_argument(dest='filenames',metavar='filename', nargs='

parser.add_argument('-p', '--pat',metavar='pattern', required=Tr
                        dest='patterns', action='append',
                        help='text pattern to search for')

parser.add_argument('-v', dest='verbose', action='store_true',
                        help='verbose mode')

parser.add_argument('-o', dest='outfile', action='store',
                        help='output file')

parser.add_argument('--speed', dest='speed', action='store',
                        choices={'slow','fast'}, default='slow',
                        help='search speed')

args = parser.parse_args()

# Output the collected arguments



print


(args.filenames)
print


(args.patterns)
print


(args.verbose)
print


(args.outfile)
print
```

```
(args.speed)
```

这个程序定义了一个命令行解析器，其用法是这样的：

```
bash % python3 search.py -h
usage: search.py [-h] [-p pattern] [-v] [-o OUTFILE] [--speed {s
                     [filename [filename ...]]

Search some files

positional arguments:
  filename

optional arguments:
  -h, --help            show this help message and exit
  -p pattern, --pat pattern
                             text pattern to search for
  -v                    verbose mode
  -o                    OUTFILE output file
  --speed {slow,fast}    search speed
```

接下来的交互式会话展示了数据在程序中的显示方式，请仔细观察print()语句的输出。

```
bash % python3 search.py foo.txt bar.txt
usage: search.py [-h] -p pattern [-v] [-o OUTFILE] [--speed {fas
                     [filename [filename ...]]
search.py: error: the following arguments are required: -p/--pat

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt
filenames  = ['foo.txt', 'bar.txt']
```

```
patterns   = ['spam', 'eggs']
verbose    = True
outfile    = None
speed      = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = results
speed     = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -
               --speed=fast
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = results
speed     = fast
```

如何对选项做进一步的处理则取决于程序员自己。只要把print()替换成其他更有意义的函数即可。

13.3.3　讨论

argparse模块是标准库中最为庞大的模块之一，有着非常多的配置选项。本节仅展示其中的基本子集，可以通过使用这些子集来入门并进行扩展。

要解析命令行选项，首先需要创建一个ArgumentParser实例，并通过使用add_argument()方

法来添加想要支持的选项声明。在每个add_argument()调用中，参数dest指定了用来保存解析结果的属性名称。而当产生帮助信息时会用到参数metavar。参数action则指定了与参数处理相关的行为，通常用store来表示存储单个值，或者用append来表示将多个值保存到一个列表中。

下面的参数表示将所有额外的命令行参数保存到一个列表中。在示例中，这条语句用来创建一个文件名列表：

```
parser.add_argument(dest='filenames',metavar='filename', nargs='
```

接下来的参数设定了一个布尔标记，标记的值取决于参数是否有提供：

```
parser.add_argument('-v', dest='verbose', action='store_true',
                        help='verbose mode')
```

下面的参数表示接受一个单独的值并将其保存为字符串：

```
parser.add_argument('-o', dest='outfile', action='store',
                        help='output file')
```

下面语句中指定的参数可允许命令行参数重复多次，并将所有的参数值保存在列表中。required标记意味着参数必须至少要提供一次。使用-p和--pat表示这两种选项名称都是可接受的。

```
parser.add_argument('-p', '--pat',metavar='pattern', required=Tru
                        dest='patterns', action='append',
                        help='text pattern to search for')
```

最后，下面语句中指定的参数可接受一个值，但会将这个值同一组可能的选择做对比。

```
parser.add_argument('--speed', dest='speed', action='store',
                        choices={'slow','fast'}, default='slow',
                        help='search speed')
```

一旦选项已经给出，只需要简单地执行parser.parse()方法。这么做会处理sys.argv的值，并返回结果实例。每个命令行参数解析出的结果都会保存在由dest参数所指定的对应的属性中。

还有其他几种方法可用来解析命令行选项。例如，我们可能会倾向于自己手动处理sys.argv或者使用getopt模块（仿照类似的C库打造）。但是，如果采用这种方法，那就会导致重复编写很多argparse已经提供的代码。也许会遇到使用optparse

库来解析命令行选项的代码。尽管optparse和argparse很相似，但后者更加现代化，在新项目中应该优先选择使用argparse。

13.4 在运行时提供密码输入提示

13.4.1 问题

我们已经编写好了一个脚本，其中需要用户输入密码。但是由于脚本是用来做交互式使用的，我们想为用户提供密码输入提示（此时用户输入的密码不会显示在终端上）而不是将其硬编码到脚本中。

13.4.2 解决方案

在这种情况下，Python的getpass模块正是你所需要的。它可以让我们非常方便地为用户提供密码输入，而不会将输入的密码显示在终端屏幕上。示例如下：

```
import getpass



user = getpass.getuser()
passwd = getpass.getpass()

if
```

```
svc_login(user, passwd): # You must write svc_login()

   print

('Yay!')
else

:
   print

('Boo!')
```

在上述代码中，函数svc_login()是我们必须自行编写的代码，用来进一步处理输入的密码。显然，具体的处理步骤是特定于应用的。

13.4.3 讨论

注意，在上面的代码中，getpass.getuser()并不会提示用户输入用户名。相反，它会根据用户的shell环境使用当前的用户登录名。或者作为最后的手段，以本地系统的密码数据库（在支持pwd模块的平台上）为支撑。

如果为了更加可靠，想显式给用户提供用户名输入，那么可以使用内置的input函数：

```
user = input('Enter your username: ')
```

同样需要记得的是，在有些系统上可能不支持将输入给getpass()方法的密码做隐藏处理。在这种情况下，Python会竭尽所能的发出预警信息（例如，在继续处理前警告你密码将以明文形式显示）。

13.5 获取终端大小

13.5.1 问题

我们需要获取终端的大小，以此对程序的输出做适当的格式化处理。

13.5.2 解决方案

可以使用os.get_terminal_size()函数来办到：

```
>>> import os



>>> sz = os.get_terminal_size()
>>> sz
os.terminal_size(columns=80, lines=24)
>>> sz.columns
80
>>> sz.lines
24
>>>
```

13.5.3 讨论

还有许多其他的方法来获取终端的大小，从读

取环境变量到执行涉及ioctl()和TTY的底层系统调用都可以。坦白地说，如果一个简单的调用就能解决问题，为什么还要操心那些细节呢？

13.6 执行外部命令并获取输出

13.6.1 问题

我们想执行一个外部命令并把输出保存为一个Python字符串。

13.6.2 解决方案

可以使用函数subprocess.check_output()来完成。示例如下：

```
import subprocess

out_bytes = subprocess.check_output(['netstat','-a'])
```

这么做会运行指定的命令，并将输出结果以字节串的形式返回。如果需要将返回的结果字节以文本的形式来解读，可以再增加一个解码的步骤。示例如下：

```
out_text = out_bytes.decode('utf-8')
```

如果执行的命令返回了一个非零的退出码，那么就会产生一个异常。下面的示例可以捕获错误并获取创建的输出以及退出码：

```
try

:
    out_bytes = subprocess.check_output(['cmd','arg1','arg2'])
except

subprocess.CalledProcessError as

e:
    out_bytes = e.output         # Output generated before error

    code = e.returncode         # Return code
```

默认情况下，check_output()只会返回写入到标准输出中的结果。如果希望标准输出和标准错误输出都能获取到，那么可以使用参数stderr：

```
out_bytes = subprocess.check_output(['cmd','arg1','arg2'],
```

```
                                          stderr=subprocess.STDOUT)
```

如果需要执行一个带有超时机制的命令，可以使用参数timeout：

```
try

:
    out_bytes = subprocess.check_output(['cmd','arg1','arg2'], ti
except

subprocess.TimeoutExpired as

e:
    ...
```

一般来说，命令的执行不需要依赖底层shell的支持（例如sh、bash等）。相反，我们提供的字符串列表会传递给底层的系统命令，比如os.execve()。如果希望命令通过shell来解释执行，只要将命令以简单的字符串形式提供并给定参数shell=True即可。如果打算让Python执行一个涉及管道、I/O重定向或其他特性的复杂shell命令时，这么做往往是很有用的。例如：

```
out_bytes = subprocess.check_output('grep python | wc > out', she
```

请注意，在shell下执行命令是有着潜在的安全威胁的，特别是当参数来自于用户的输入时更是如此。在这种情况下，shlex.quote()函数可用来正确引用包含在shell命令中的参数。

13.6.3 讨论

执行一个外部命令并获取输出，最简单的方法就是使用check_output()函数了。但是，如果需要同一个子进程执行更加高级的通信，例如为其发送输入，那就需要采用不同的方法了。基于此，可以直接使用subprocess.Popen类。示例如下：

```
import subprocess

# Some text to send

text = b'''
hello world
this is a test
goodbye
'''

# Launch a command with pipes
```

```
p = subprocess.Popen(['wc'],
          stdout = subprocess.PIPE,
          stdin = subprocess.PIPE)

# Send the data and get the output



stdout, stderr = p.communicate(text)

# To interpret as text, decode



out = stdout.decode('utf-8')
err = stderr.decode('utf-8')
```

如果某个外部命令期望同一个真正的TTY（即，终端设备）进行交互，那么subprocess模块不适合同这样的外部命令进行通信。例如，我们不能用它来实现自动向用户请求输入密码的任务（比如一个ssh会话）。对于这种需求，需要使用第三方模块来完成，比如那些基于流行的“expect”族的工具（如pexpect或类似的工具）。

13.7 拷贝或移动文件和目录

13.7.1 问题

我们需要拷贝或移动文件和目录，但是不想通过调用shell命令来完成。

13.7.2 解决方案

shutil模块中有着可移植的函数实现，可用来拷贝文件和目录，用法相当直接，示例如下：

```
import shutil



# Copy src to dst. (cp src dst)


shutil.copy(src, dst)

# Copy files, but preserve metadata (cp -p src dst)


shutil.copy2(src, dst)

# Copy directory tree (cp -R src dst)
```

```
shutil.copytree(src, dst)

# Move src to dst (mv src dst)


shutil.move(src, dst)
```

这些函数的参数全都是字符串，用来提供文件或目录的名称。如同注释中说明的那样，这些函数的底层语义是在尝试模仿类似的UNIX命令。

默认情况下，符号链接也适用于这些命令。例如，如果源文件是一个符号链接，那么目的文件将会是该链接所指向的文件的拷贝。如果只想拷贝符号链接本身，可以提供关键字参数follow_symlinks，示例如下：

```
shutil.copy2(src, dst, follow_symlinks=False)
```

如果想在拷贝的目录中保留符号链接，可以这么做：

```
shutil.copytree(src, dst, symlinks=True)
```

copytree()函数以可选的方式允许在拷贝的过程中忽略特定的文件和目录。为了做到这点，需要提供一个ignore函数，该函数以目录名和文件名作为输入参数，返回一列要忽略的名称作为结果。示例如下：

```
def

ignore_pyc_files(dirname, filenames):
    return

[name in

filenames if

name.endswith('.pyc')]

shutil.copytree(src, dst, ignore=ignore_pyc_files)
```

由于忽略文件名这种模式非常常见，已经有一个实用函数ignore_patterns()提供给我们使用了。示例如下：

```
shutil.copytree(src, dst, ignore=shutil.ignore_patterns('*~','*.
```

13.7.3 讨论

大部分情况下用shutil来拷贝文件和目录都是非常直接的。但是，需要注意的是，当考虑到文件的元数据时，类似copy2()这样的函数只会尽最大努力来保存这些数据。一些基本信息像访问时间、创建时间以及权限信息总是会得到保存。但是属主、访问控制列表、资源派生（resource forks）以及其他扩展的文件元数据可能会也可能不会得到保存，这取决于操作系统底层和用户自身的访问权限。你很可能不会想去用shutil.copytree()这样的函数来做系统备份。

当和文件名打交道时，确保使用的是os.path中的函数，这样可获得最佳的可移植性（尤其是如果需要同时运行于UNIX和Windows上时）。例如：

```
>>> filename = '/Users/guido/programs/spam.py'
>>> import os.path



>>> os.path.basename(filename)
'spam.py'
>>> os.path.dirname(filename)
'/Users/guido/programs'
>>> os.path.split(filename)
('/Users/guido/programs', 'spam.py')
>>> os.path.join('/new/dir', os.path.basename(filename))
'/new/dir/spam.py'
>>> os.path.expanduser('~/guido/programs/spam.py')
'/Users/guido/programs/spam.py'
>>>
```

用copytree()来拷贝目录时，一个比较棘手的点在于错误处理。比如说，在拷贝过程中函数可能会遇到已经损坏的符号链接，或者由于权限问题导致有些文件无法访问等等诸如此类的问题。为了应对这些情况，所有遇到的异常都会收集到一个列表中并将其归组为一个单独的异常，在操作结束时抛出。示例如下：

```
try

:
    shutil.copytree(src, dst)
except

shutil.Error as

e:
    for

src, dst, msg in

e.args[0]:
         # src is source name


         # dst is destination name
```

```
        # msg is error message from exception
        print
(dst, src, msg)
```

如果提供了关键字参数ignore_dangling_sumlinks=True，那么copytree()将会忽略悬垂的符号链接。

本节中展示的函数可能是最常使用的了。但是，shutil模块中还有许多有关拷贝数据的操作。进一步阅读相关的文档肯定是值得的，参见 http://docs.python.org/3/library/shutil.html 。

13.8　创建和解包归档文件

13.8.1　问题

我们需要以常见的格式（如.tar、.tgz或.zip）来创建或解包归档文件。

13.8.2　解决方案

shutil模块中有两个函数——make_archive()和unpack_archive()，它们正是我们所需要的。示例如下：

```
>>> import shutil



>>> shutil.unpack_archive('Python-3.3.0.tgz')
>>> shutil.make_archive('py33','zip','Python-3.3.0')
'/Users/beazley/Downloads/py33.zip'
>>>
```

make_archive()的第二个参数就是所期望的输出格式。要获取所支持的归档格式列表，可以使用get_archive_formats()函数。示例如下：

```
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"), ('gztar', "gzip'ed tar-file"),
('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]
>>>
```

13.8.3 讨论

Python中还有其他模块可用来处理各种归档格式的底层细节（例如tarfile、zipfile、gzip、bz2等）。但是，如果想做的只是创建或解包归档文件，那么确实没有必要使用如此底层的模块。可以直接使用shutil模块中的高层函数来解决。

这些函数有着许多额外的选项，可用于记录日志、文件权限等。可参阅shutil模块的文档以获得更多的细节。

13.9　通过名称来查找文件

13.9.1　问题

我们需要编写一个涉及查找文件的脚本，比如给文件重命名或者日志归档程序。但是我们不想在Python脚本中调用shell实用程序，也不想提供特定的行为使得程序无法轻易地分发出去使用。

13.9.2　解决方案

搜索文件可使用os.walk()函数，只要将顶层目录提供给它即可。下面示例中给出的函数用来查找一个特定的文件名，并将所有匹配结果的绝对路径打印出来：

```
#!/usr/bin/env python3.3

import os

def
findfile(start, name):
```

```
    for

relpath, dirs, files in

os.walk(start):
        if

name in

files:
            full_path = os.path.join(start, relpath, name)
            print

(os.path.normpath(os.path.abspath(full_path)))

if

__name__ == '__main__':
    findfile(sys.argv[1], sys.argv[2])
```

将这个脚本保存为findfile.py并从命令行运行，给定查找的起始点以及待匹配的文件名，就像下面这样：

```
bash % ./findfile.py . myfile.txt
```

13.9.3　讨论

os.walk()方法会为我们遍历目录层级，且对于进入的每个目录它都会返回一个3元组。这包含了正在检视的目录的相对路径、该目录中包含的所有目录名的列表，以及该目录中包含的所有文件名的列表。

对于每个元组，只需检查目标文件是否在file列表中即可。如果是，就用os.path.join()来组成一个路径。为了避免出现可能像././foo//bar这样的诡异路径，还要用到两个额外的函数来修正结果。第一个是os.path.abspath()，它接受一个可能是相对的路径并将其组成绝对路径形式。第二个是os.path.normpath()，它会将路径修正为标准化形式，从而帮助解决像双反斜线、多当前目录的多次引用等问题。

尽管这个脚本同UNIX平台上的find实用程序相比显得异常简单，但它具有跨平台的优势。此外，许多额外的功能都能够以可移植的方式加入进来而无需耗费太多精力。为了说明这一点，下面这个函数可打印出所有最近有修改过的文件：

```
#!/usr/bin/env python3.3



import os
```

```
import time



def


modified_within(top, seconds):
    now = time.time()
    for


path, dirs, files in


os.walk(top):
        for


name in


files:
            fullpath = os.path.join(path, name)
            if


os.path.exists(fullpath):
                mtime = os.path.getmtime(fullpath)
                if


mtime > (now - seconds):
                    print


(fullpath)

if
```

```
__name__ == '__main__':
    import sys



    if


len(sys.argv) != 3:
        print


('Usage: {} dir seconds'.format(sys.argv[0]))
        raise SystemExit


(1)

modified_within(sys.argv[1], float(sys.argv[2]))
```

在这个短小的函数之上构建更加复杂的操作并不会花费太多时间，只要使用os、os.path、glob以及类似模块中的各种功能即可。相关章节可参阅5.11节和5.13节。

13.10 读取配置文件

13.10.1 问题

我们想要读取以常见的.ini格式所编写的配置文件。

13.10.2 解决方案

可以用configparser模块来读取配置文件。例如，假设有一个这样的配置文件：

```
; config.ini
; Sample configuration file

[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local

# Setting related to debug configuration
[debug]
log_errors=true
show_warnings=False

[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
signature:
    =================================
```

```
    Brought to you by the Python Cookbook
    =================================
```

下面的示例告诉我们如何读取这个配置文件并提取出相应的值：

```
>>> from configparser import


ConfigParser
>>> cfg = ConfigParser()
>>> cfg.read('config.ini')
['config.ini']
>>> cfg.sections()
['installation', 'debug', 'server']
>>> cfg.get('installation','library')
'/usr/local/lib'
>>> cfg.getboolean('debug','log_errors')
True
>>> cfg.getint('server','port')
8080
>>> cfg.getint('server','nworkers')
32
>>> print


(cfg.get('server','signature'))

=================================
Brought to you by the Python Cookbook
=================================
>>>
```

如果需要，也可以使用cfg.write()方法修改配置并写回到原文件中。示例如下：

```
>>> cfg.set('server','port','9000')
>>> cfg.set('debug','log_errors','False')
>>> import sys



>>> cfg.write(sys.stdout)
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
signature =
          =================================
          Brought to you by the Python Cookbook
          =================================
>>>
```

13.10.3　讨论

配置文件以易于人类阅读的格式对程序设定配置数据。在每个配置文件中，值被归组到不同的区段中（例如本例中的“installation”、“debug”和“server”），然后在每个区段中对各个变量设定值。

在配置文件和使用Python来编写的用于同样目的的源文件之间有着几个显著的区别。首先，前者的语法更加宽容和“草率”。例如，下面这些赋值语句的效果相同：

```
prefix=/usr/local
prefix: /usr/local
```

在配置文件中用到的名称也被认为是非大小写敏感的。例如：

```
>>> cfg.get('installation','PREFIX')
'/usr/local'
>>> cfg.get('installation','prefix')
'/usr/local'
>>>
```

当解析值的时候，像getboolean()这样的方法会检查任何合理的值。例如，下面这些语句的效果都是相同的：

```
log_errors = true
log_errors = TRUE
log_errors = Yes
log_errors = 1
```

和脚本不同，也许在配置文件和Python代码之

间最大的区别在于配置文件不是按照从上到下的方式来执行的。相反，配置文件会被全部读取。如果其中出现变量替换的操作，则它们都是在文件全部读取之后才进行的。比如说在如下部分中，在其他变量使用prefix之前是否对它完成了赋值是无关紧要的。

```
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
```

关于ConfigParser，一个容易忽视的特性是它可以分别读取多个配置文件并将它们的结果合并成一个单独的配置。例如，假设某位用户创建了他们自己的配置文件，看起来是这样的：

```
; ~/.config.ini
[installation]
prefix=/Users/beazley/test

[debug]
log_errors=False
```

这个文件可以单独读取，再同前面的配置合并在一起。示例如下：

```
>>> # Previously read configuration



>>> cfg.get('installation', 'prefix')
'/usr/local'

>>> # Merge in user-specific configuration



>>> import os



>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']
>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
>>>
```

注意观察对变量prefix的修改是如何影响到其他相关的变量的，比如对library的设定。这么做行得通是因为对变量的插值操作是尽可能晚才执行的。可以通过下面的实验看出这一点：

```
>>> cfg.get('installation','library')
'/Users/beazley/test/lib'
>>> cfg.set('installation','prefix','/tmp/dir')
>>> cfg.get('installation','library')
'/tmp/dir/lib'
>>>
```

最后，需要重点提到的是Python并不能对在其他程序中使用的.ini文件的全部特性都提供支持（例如Windows上的应用）。确保参考configparser的文档以获得更详细的语法和所支持特性的细节。

13.11 给脚本添加日志记录

13.11.1 问题

我们想让脚本和简单的程序可以将诊断信息写入到日志文件中。

13.11.2 解决方案

给简单的程序添加日志功能，最简单方法就是使用logging模块了。示例如下：

```
import logging



def


main():
    # Configure the logging system



    logging.basicConfig(
        filename='app.log',
        level=logging.ERROR
    )

    # Variables (to make the calls that follow work)
```

```
    hostname = 'www.python.org'
    item = 'spam'
    filename = 'data.csv'
    mode = 'r'
    # Example logging calls (insert into your program)



    logging.critical('Host %s unknown', hostname)
    logging.error("Couldn't find %r", item)
    logging.warning('Feature is deprecated')
    logging.info('Opening file %r, mode=%r', filename, mode)
    logging.debug('Got here')

if


__name__ == '__main__':
    main()
```

这5个logging调用（critical()、error()、warning()、info()、debug()）分别代表着不同的严重级别，以降序排列。basicConfig()的level参数是一个过滤器，所有等级低于此设定的消息都会被忽略掉。

每个日志操作的参数都是一条字符串消息，后面跟着零个或多个参数。当产生日志消息时，%操作符使用提供的参数来格式化字符串消息。

如果运行这个程序，文件*app.log* 的内容将会

是这样的：

```
CRITICAL:root:Host www.python.org unknown
ERROR:root:Could not find 'spam'
```

如果想改变输出或输出的严重级别，可以通过修改调用basicConfig()的参数来实现。示例如下：

```
logging.basicConfig(
     filename='app.log',
     level=logging.WARNING,
     format='%(levelname)s:%(asctime)s:%(message)s')
```

修改后输出结果变成了下面这样：

```
CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown
ERROR:2012-11-20 12:27:13,595:Could not find 'spam'
WARNING:2012-11-20 12:27:13,595:Feature is deprecated
```

如上所示，日志的配置信息被直接硬编码到了程序中。如果想从配置文件中进行配置，把basicConfig()调用修改成如下形式：

```
import logging
```

```
import logging.config



def


main():
    # Configure the logging system



    logging.config.fileConfig('logconfig.ini')
    ...
```

现在创建一个配置文件*logconfig.ini*，看起来是这样的：

```
[loggers]
keys=root

[handlers]
keys=defaultHandler

[formatters]
keys=defaultFormatter

[logger_root]
level=INFO
handlers=defaultHandler
qualname=root

[handler_defaultHandler]
class=FileHandler
formatter=defaultFormatter
args=('app.log', 'a')

[formatter_defaultFormatter]
```

```
format=%(levelname)s:%(name)s:%(message)s
```

如果想修改配置，直接编辑*logconfig.ini* 文件即可。

13.11.3　讨论

对于logging模块来说有着上百万种高级的配置选项可用，但此时让我们暂时忽略这些细节。本节给出的解决方案对于简单的程序和脚本来说已经足够用了。只要保证在调用任何logging调用之前先调用basicConfig()即可，这样你的程序将能够产生日志输出。

如果想让日志消息发送到标准错误输出而不是文件中，不要给basicConfig()提供任何文件名做参数即可。例如，可以这么做：

```
logging.basicConfig(level=logging.INFO)
```

关于basicConfig()，一个微妙的地方在于它只能在程序中调用一次。如果稍后需要修改日志模块的配置，需要取得根日志对象（root logger）并直接对其做修改。示例如下：

```
logging.getLogger().level = logging.DEBUG
```

必须要强调的是，本节只展示了logging模块的几个基本用法，还有相当多的高级定制化操作可做。对于这样的定制化操作，一个极佳的资源是“Logging Cookbook”（http://docs.python.org/3/howto/logging-cookbook.html ）。

13.12 给库添加日志记录

13.12.1 问题

我们想给一个库添加日志功能，但是又不希望它影响那些没有使用日志功能的程序。

13.12.2 解决方案

对于想执行日志记录的库来说，应该创建一个专用的日志对象并将其初始化为如下形式：

```
# somelib.py

import logging

log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

# Example function (for testing)

def

func():
```

```
    log.critical('A Critical Error!')
    log.debug('A debug message')
```

有了这样的配置，默认情况下将不会产生任何日志输出。例如：

```
>>> import somelib
>>> somelib.func()
>>>
```

但是，如果日志系统得到适当的配置，则日志消息将开始出现。示例如下：

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

13.12.3　讨论

库给日志带来了一个特殊的问题：即，使用日志的环境是未知的。一般来说，绝不应该在库代码中尝试去自行配置日志系统，或者对已有的日志配置做任何假设。因此，需要小心翼翼地提供隔离措施。

getLogger(__name__)创建了一个日志模块，其名称同调用它的模块名相同。由于所有的模块都是唯一的，这么做就创建了一个专用的日志对象，也就与其他的日志对象隔离开了。

log.addHandler(logging.NullHandler())操作绑定了一个空的处理例程到刚刚创建的日志对象上。默认情况下，空处理例程会忽略所有的日志消息。因此，如果用到了这个库且日志系统从未配置过，那么就不会出现任何日志消息或警告信息。

对单个库的日志记录可以独立地进行配置，不必管其他的日志设定。例如，考虑如下的代码：

```
>>> import logging



>>> logging.basicConfig(level=logging.ERROR)
>>> import somelib



>>> somelib.func()
CRITICAL:somelib:A Critical Error!
```

```
>>> # Change the logging level for 'somelib' only



>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
>>>
```

这里，根日志对象已经被配置为只输出ERROR或更高等级的消息。但是，somelib库的日志等级已经被单独配置为输出调试消息了，这个设定的优先级要高于全局设定。

对于单个模块来说，能够像这样修改日志的设定会是一个非常有用的调试工具。因为我们不必去修改任何全局的日志设定了——当某个模块需要更多的日志输出时，只要针对这个模块修改日志等级即可。

“Logging HOWTO”（http://docs.python.org/3/howto/logging.htm）一文中有关于配置logging模块以及其他一些有用技巧的更多信息。

13.13 创建一个秒表计时器

13.13.1 问题

我们想记录执行各项任务所花费的时间。

13.13.2 解决方案

time模块中包含了各种与计时相关的函数。但是，通常在这些函数之上构建更高层的接口来模拟秒表会更有用。示例如下：

```
import time



class Timer


:
    def


__init__(self, func=time.perf_counter):
        self.elapsed = 0.0
        self._func = func
        self._start = None

    def


```

```
start(self):
        if


self._start is not


None:
            raise RuntimeError


('Already started')
        self._start = self._func()

    def


stop(self):
        if


self._start is


None:
            raise RuntimeError


('Not started')
        end = self._func()
        self.elapsed += end - self._start
        self._start = None

    def


reset(self):
        self.elapsed = 0.0

@property
def


running(self):
    return
```

```
self._start is not

None

def

__enter__(self):
    self.start()
    return

self

def

__exit__(self, *args):
    self.stop()
```

这个类定义了一个定时器，可以根据用户的需要启动、停止和重置它。Timer类将总的花费时间记录在elapsed属性中。下面的示例展示了如何使用这个类：

```
def

countdown(n):
    while

n > 0:
```

```
        n -= 1

# Use 1: Explicit start/stop



t = Timer()
t.start()
countdown(1000000)
t.stop()
print


(t.elapsed)

# Use 2: As a context manager



with


t:
    countdown(1000000)
print


(t.elapsed)

with


Timer() as


t2:
    countdown(1000000)
print


(t2.elapsed)
```

13.13.3 讨论

本节提供了一个简单但非常有用的类，可用来进行计时和跟踪花费的时间。这个类也很好地演示了如何支持上下文管理协议以及对with语句的使用。

在进行计时测量时需要考虑底层所用到的时间函数。一般来说，像time.time()或者time.clock()的计时精度根据操作系统的不同而有所区别。相反，time.perf_counter()函数总是会使用系统中精度最高的计时器。

如同前面所展示的，由Timer类记录的时间是系统时钟时间，其中包含了所有的休眠期时间。如果只想获取进程的CPU时间，可以用time.process_time()取代。示例如下：

```
t = Timer(time.process_time)
with t:
    countdown(1000000)
print(t.elapsed)
```

time.perf_counter()和time.process_time()都返回秒级的时间值（以浮点数表示）。但是单独这样一个时间值没有任何意义，要使得结果变得有意义，必须调用函数两次并计算两次时间的差。

有关计时和性能统计分析的更多主题请参阅14.13节。

13.14　给内存和CPU使用量设定限制

13.14.1　问题

我们想对运行在UNIX系统上的程序在内存和CPU的使用量上设定一些限制。

13.14.2　解决方案

resource模块可用来执行这样的任务。例如，要限制CPU时间可以这样做：

```
import signal

import resource

import os


def

time_exceeded(signo, frame):
```

```
    print

("Time's up!")
    raise SystemExit

(1)

def

set_max_runtime(seconds):
    # Install the signal handler and set a resource limit


    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
    signal.signal(signal.SIGXCPU, time_exceeded)

if

__name__ == '__main__':
    set_max_runtime(15)
    while

True:
        pass
```

运行上述代码，当超时时会产生SIGXCPU信号。程序就会做清理工作然后退出。

要限制内存的使用，可以在使用的总地址空间上设定一个限制。示例如下：

```
import resource



def

limit_memory(maxsize):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (maxsize, hard))
```

当设定了内存限制后，如果没有更多的内存可用，程序就会开始产生MemoryError异常。

13.14.3　讨论

在本节中，我们通过setrlimit()函数来为特定的资源设定软性和硬性限制。软性限制就是一个值，一般来说操作系统会通过信号机制来限制或通知进程。硬性限制代表着软性限制值的上限。通常，这些值是由系统全局参数所控制的，它们会由系统管理员来设定。虽然可以降低硬性限制，但这个过程不能由用户进程来控制（就算这个进程要降低自己的硬性限制也不行）。

setrlimit()函数还可以用来设定比如子进程数量、可打开的文件数量等系统资源。可以参阅

resource模块的文档以获得进一步的细节。

请注意，本节中的技术只能用于UNIX系统，而且可能并不适用于所有的UNIX变种。例如，当我们进行测试时发现在Linux上可以正常工作但在OS X上就不行。

13.15　加载Web浏览器

13.15.1　问题

我们想从脚本中加载一个浏览器并让它打开指定的URL。

13.15.2　解决方案

webbrowser模块可用来以独立于平台的方式加载浏览器。示例如下：

```
>>> import webbrowser



>>> webbrowser.open('http://www.python.org')
True
>>>
```

这会用默认的浏览器打开请求的页面。如果想对页面打开的方式有更多的控制，可以使用下列函数之一：

```
>>> # Open the page in a new browser window
```

```
>>> webbrowser.open_new('http://www.python.org')
True
>>>

>>> # Open the page in a new browser tab



>>> webbrowser.open_new_tab('http://www.python.org')
True
>>>
```

如果浏览器支持的话，这么做会尝试在一个新的浏览器窗口或标签页中打开页面。

如果想在特定的浏览器中打开页面，可以使用webbrowser.get()函数来指定一个具体的浏览器。示例如下：

```
>>> c = webbrowser.get('firefox')
>>> c.open('http://www.python.org')
True
>>> c.open_new_tab('http://docs.python.org')
True
>>>
```

支持的浏览器名称的完整列表可以在 Python 文档（http://docs.python.org/3/library/webbrowser.html ）中找到。

13.15.3　讨论

在许多脚本中，能够轻松加载一个浏览器可算是一项有用的操作。例如，也许脚本执行了某种服务器部署的任务，而我们想马上加载浏览器来验证这是否能够工作。或者某个程序把数据以HTML页面的形式输出，而我们只想打开浏览器查看结果。无论怎样，webbrowser模块都是一种简单的解决方案。

第14章　测试、调试以及异常

测试是很棒的一件事，但调试就没那么有趣了吧。在Python解释器执行代码之前并没有编译器来分析你的代码，这一事实使得测试成为了开发中至关重要的部分。本章的目的是讨论一些与测试、调试以及异常处理相关的常见问题。本章不是为测试驱动开发（TDD）或者unittest模块做简要的介绍。因此，我们假设读者已经对软件测试方面的一些概念有所了解。

14.1　测试发送到stdout上的输出

14.1.1　问题

我们的程序中有一个方法会将输出发送到标准输出上（sys.stdout）。这几乎总是表示它会把文本发送到屏幕上。我们想为自己的代码编写一个测试用例，以此证明只要给定合适的输入，则会在屏幕上显示合适的输出。

14.1.2　解决方案

利用unittest.mock模块的patch()函数，很容易为单独的测试用例模拟出sys.stdout。用完后可将其放回，不必使用临时变量或者在测试用例之间暴露出模拟的状态。

考虑下面这个位于mymodule模块中的函数：

```
# mymodule.py

def urlprint(protocol, host, domain):
    url = '{}://{}.{}'.format(protocol, host, domain)
    print(url)
```

内建的print函数默认情况下会把输出发往sys.stdout。为了测试输出确实会发往sys.stdout，可以利用一个对象作为sys.stdout的替身来模拟这种情况，然后对产生的结果做断言处理。利用unittest.mock模块的patch()方法能够非常方便地替换对象，而且只在运行的测试用例的上下文中生效。在测试完成后，会立刻将所有的东西返回到它们的原始状态上。下面就是针对mymodule的测试代码：

```
from io import StringIO
from unittest import TestCase
from unittest.mock import patch
import mymodule

class TestURLPrint(TestCase):
    def test_url_gets_to_stdout(self):
        protocol = 'http'
        host = 'www'
        domain = 'example.com'
        expected_url = '{}://{}.{}\n'.format(protocol, host, doma

        with patch('sys.stdout', new=StringIO()) as fake_out:
            mymodule.urlprint(protocol, host, domain)
            self.assertEqual(fake_out.getvalue(), expected_url)
```

14.1.3　讨论

urlprint()函数接受三个参数，测试代码首先为每个参数设定一个哑值（dummy value）。变量expected_url被设定为包含期望输出的字符串。

要运行这个测试用例，unittest.mock.patch()函数用来当做上下文管理器，把sys.stdout的值替换为一个StringIO对象。在这个过程中会创建一个模拟对象，即fake_out变量。可以在with语句块中使用fake_out来执行各种检查。当with语句块执行完毕后，patch()函数会非常方便地将所有状态还原为测试运行之前时的状态。

值得一提的是，某些特定的C扩展模块可能会通过设定sys.stdout直接往标准输出写数据。本节不针对这种情况做特别说明，但对于纯Python代码来说应该是能正常工作的（如果需要从这种C扩展模块中捕获I/O，可以打开一个临时文件，然后让标准输出临时重定向到那个文件来完成。这中间涉及各种操作文件描述符的技巧）。

有关在字符串和StringIO对象中捕获I/O的更多信息可参见5.6节。

14.2 在单元测试中为对象打补丁

14.2.1 问题

我们正在编写单元测试，需要对选定的对象添加补丁，以此才能在测试中针对它们的使用情况做断言处理（例如，对特定的调用参数做断言、访问选定的属性时做断言等）。

14.2.2 解决方案

unittest.mock.patch()函数可用来帮助解决这个问题。虽然不太常见，但patch()函数用法较多，可当做装饰器、上下文管理器或者单独使用。例如，在下面的示例中我们把它当做装饰器来用：

```
from unittest.mock import patch
import example

@patch('example.func')
def test1(x, mock_func):
    example.func(x)         # Uses patched example.func
    mock_func.assert_called_with(x)
```

也可以把它当做上下文管理器来用：

```
with patch('example.func') as mock_func:
    example.func(x)         # Uses patched example.func
    mock_func.assert_called_with(x)
```

最后但同样重要的是，也可以用它来手动打补丁：

```
p = patch('example.func')
mock_func = p.start()
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

如果有必要的话，可以将装饰器和上下文管理器堆叠起来对多个对象打补丁。示例如下：

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1, \
         patch('example.patch2') as mock2, \
         patch('example.patch3') as mock3:
    ...
```

14.2.3 讨论

patch()接受一个已有对象的完全限定名称并将其替换为一个新值。在装饰器函数或者上下文管理器结束执行后会将对象恢复为原始值。默认情况下，对象会被替换为MagicMock实例。示例如下：

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

但是，实际上可以将对象替换为任何你希望的值，只要将值作为patch()的第二个参数传入即可：

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

MagicMock实例一般被当作替换值来用，旨在模仿可调用对象和实例。它们会记录使用信息而且允许创建断言。示例如下：

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>

>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called

>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>

>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

一般来说，这类操作都是在单元测试中进行的。例如，假设有如下的函数：

```
# example.py
from urllib.request import urlopen
import csv

def dowprices():
```

```
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices
```

通常，这个函数会使用urlopen()从Web上抓取一些数据然后进行解析。要对这个函数做单元测试，我们可能会想自己创建一份更加可预测的数据集，然后将其作为函数的测试数据。下面的示例采用了前文讨论过的补丁技术：

```
import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
                         {'IBM': 91.1,
                          'AA': 13.25,
                          'MSFT' : 27.72})

if __name__ == '__main__':
    unittest.main()
```

在这个示例中，example模块中的urlopen()函数被替换成了一个mock对象，返回的BytesIO()中就包含着作为替代的样例数据。

关于这个测试，一个重要但微妙的地方在于我们是对example.urlopen打补丁而不是针对urllib.request.urlopen。在打补丁时，使用的名称必须和被测代码中使用的名称保持一致。由于示例代码使用的是from urllib.request import urlopen，因此实际上由函数dowprices()调用的urlopen()其实是位于example模块中的。

本节仅仅只是小小体验了一下unittest.mock模块的功能。要使用更加高级的功能和特性，官方文档（http://docs.python.org/3/library/unittest.mock ）是必读的资料。

14.3 在单元测试中检测异常情况

14.3.1 问题

我们想编写一个能够快速检测异常的单元测试。

14.3.2 解决方案

检测异常可使用assertRaise()方法。例如，如果想检查某个函数是否引发了ValueError异常，可以使用下面的代码完成：

```
import unittest



# A simple function to illustrate


def


parse_int(s):
    return


int(s)

```

```
class TestConversion


(unittest.TestCase):
    def


test_bad_int(self):
        self.assertRaises(ValueError


, parse_int, 'N/A')
```

如果需要以某种方式检查异常的值，那就需要用到另一种不同的方法。示例如下：

```
import errno




class TestIO


(unittest.TestCase):
    def


test_file_not_found(self):
        try


:
            f = open('/file/not/found')
        except IOError as


e:
```

```
            self.assertEqual(e.errno, errno.ENOENT)
        else

:
            self.fail('IOError not raised')
```

14.3.3 讨论

assertRaise()方法提供了一种简便的方式来测试是否有异常出现。编写测试代码时，一个常见的误区就是自己手工尝试用try和except来处理异常。比如：

```
class TestConversion


(unittest.TestCase):
    def


test_bad_int(self):
        try


:
            r = parse_int('N/A')
        except ValueError as


e:
            self.assertEqual(type(e), ValueError


)
```

这种方法的问题在于容易忘记边界情况，比如当根本没有异常产生时。为了应对这点，需要为这种情况添加一个检查，示例如下：

```
class TestConversion

(unittest.TestCase):
    def

test_bad_int(self):
        try

:
            r = parse_int('N/A')
        except ValueError as

e:
            self.assertEqual(type(e), ValueError

)
        else

:
            self.fail('ValueError not raised')
```

assertRaises()方法则替我们处理了所有这些细

节，所以应该尽量多使用它。

assertRaises()的局限性在于对于所产生的异常对象的值，并不提供测试方法。要实现这一目的，必须像上面的示例那样手动进行测试。在这两种极端情况之间，可能会考虑使用assertRaisesRegex()方法。该方法允许我们在测试异常的同时还可以针对异常的字符串表示进行正则表达式匹配。示例如下：

```
class TestConversion


(unittest.TestCase):
    def


test_bad_int(self):
        self.assertRaisesRegex(ValueError


, 'invalid literal .*',
                                       parse_int, 'N/A')
```

关于assertRaises()和assertRaisesRegex()还有一个鲜为人知的事实，即，它们也可以当做上下文管理器来使用：

```
class TestConversion
```

```
(unittest.TestCase):
    def


test_bad_int(self):
        with


self.assertRaisesRegex(ValueError


, 'invalid literal .*'):
            r = parse_int('N/A')
```

如果我们的测试除了要执行一个可调用对象之外还涉及多个步骤，那么上下文管理器的形式就很有用了。

14.4　将测试结果作为日志记录到文件中

14.4.1　问题

我们想把单元测试的结果写入到文件中，而不是打印到标准输出上。

14.4.2　解决方案

运行单元测试的一种非常常用的技术就是在测试文件底部包含下列代码：

```
import unittest



class MyTest


(unittest.TestCase):
    ...

if


__name__ == '__main__':
    unittest.main()
```

这么做会让测试文件变为可执行文件，而且会把测试结果打印到标准输出上。如果想对输出做重定向，需要将原来的main()展开，然后编写自己的main()函数。示例如下：

```
import sys

def

main(out=sys.stderr, verbosity=2):
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromModule(sys.modules[__name__])
    unittest.TextTestRunner(out,verbosity=verbosity).run(suite)

if

__name__ == '__main__':
    with

open('testing.out', 'w') as

f:
        main(f)
```

14.4.3　讨论

本节中有趣的地方不在于将测试结果重定向到文件中，而是当这么做的时候暴露了unittest模块内部值得注意的一些工作原理。

从基本的层次来说，unittest模块首先会组装一个测试套件。这个测试套件中包含了各种定义的测试方法。一旦套件装配完成，它所包含的测试就开始执行。

单元测试的这两个部分是彼此相分离的。解决方案中创建的unittest.TestLoader实例是用来组装测试套件的。而loadTestsFromModule()是TestLoader的几个实例方法之一，用来收集测试。在这种情况下，它会为TestCase类扫描模块，并从模块中提取出测试方法。如果想获得更细粒度的控制，可以用loadTestsFromTestCase()方法（本节未给出）从继承自TestCase的子类中提取测试方法。

TextTestRunner类是一个测试运行类的例子。该类的主要目的就是运行包含在测试套件中的测试。这个类也是unittest.main()函数所使用的测试运行类。但是，这里我们还对它做了一点底层配置，包括设置输出文件和输出信息的详细程度。

尽管本节中只包含了几行代码，但对于读者今后应该如何定制化unittest框架带来了一些启示。要

定制化测试套件的装配过程，可以利用TestLoader类的各种操作来完成。要定制化测试的执行，可以创建自定义的测试运行类，以此模拟出TextTestRunner类的功能。这些主题都超出了本节可以涵盖的范围。但是，unittest模块的文档中对这些底层的协议有着详尽的说明。

14.5　跳过测试，或者预计测试结果为失败

14.5.1　问题

我们想在自己的单元测试中跳过某些测试，或者选择几个测试将它们标记为预测会失败。

14.5.2　解决方案

unittest模块中有一些装饰器可作用于所选的测试方法上，以此控制它们的处理行为。示例如下：

```
import unittest

import os

import platform


class Tests

(unittest.TestCase):
```

```
    def


test_0(self):
        self.assertTrue(True)

    @unittest.skip('skipped test')
    def


test_1(self):
        self.fail('should have failed!')

    @unittest.skipIf(os.name=='posix', 'Not supported on Unix')
    def


test_2(self):
        import winreg

    @unittest.skipUnless(platform.system() == 'Darwin', 'Mac spec
    def


test_3(self):
        self.assertTrue(True)

    @unittest.expectedFailure
    def


test_4(self):
        self.assertEqual(2+2, 5)

if


__name__ == '__main__':
    unittest.main()
```

如果在一台Mac电脑上运行上述代码，将得到如下输出：

```
bash % python3 testsample.py -v
test_0 (__main__.Tests) ... ok
test_1 (__main__.Tests) ... skipped 'skipped test'
test_2 (__main__.Tests) ... skipped 'Not supported on Unix'
test_3 (__main__.Tests) ... ok
test_4 (__main__.Tests) ... expected failure

-----------------------------------------------------------------
Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)
```

14.5.3 讨论

装饰器skip()可用来跳过某个根本就不想运行的测试。skipIf()和skipUnless()在编写那些只针对特定平台或Python版本甚至其他依赖的测试时非常有用。对于已知会失败的测试项，而又不想让测试框架生成更多报告信息，那么可以使用装饰器@expectedFailure对其进行标注。

用来跳过检查的装饰器同样可以作用到整个测试类上。示例如下：

```
@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specifi
class DarwinTests
```

```
(unittest.TestCase):
...
```

14.6 处理多个异常

14.6.1 问题

我们有一段代码可以抛出几个不同的异常，而我们需要负责处理所有可能会发生的异常。要求处理的时候无需创建重复代码或者冗长的代码段。

14.6.2 解决方案

如果能够用一个单独的代码块处理所有不同的异常，可以将它们归组到一个元组中。示例如下：

```
try

:
    client_obj.get_url(url)
except

(URLError, ValueError

, SocketTimeout):
    client_obj.remove_url(url)
```

在上面的代码中，如果列出的这些异常中有任何一个出现，则会调用remove_url()方法。另一方面，如果需要对其中某个异常采取不同的处理办法，可以将其放入单独的except子句中去：

```
try

:
    client_obj.get_url(url)
except

(URLError, ValueError

):
    client_obj.remove_url(url)
except

SocketTimeout:
    client_obj.handle_url_timeout(url)
```

有许多异常都会被归组为继承体系。对于这样的异常，可以通过指定一个基类来捕获所有的异常。例如，与其像这样编写代码：

```
try

:
```

```
    f = open(filename)
except


(FileNotFoundError, PermissionError):
    ...
```

不如像这样重写except语句：

```
try


:
    f = open(filename)
except OSError


:
    ...
```

这么做是可行的，因为OSError是基类，FileNotFoundError和PermissionError异常都是它的子类。

14.6.3 讨论

值得一提的是，可以在抛出的异常上使用关键字as，尽管这并非是特定于处理多个异常的技术。

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('File not found')
    elif e.errno == errno.EACCES:
        logger.error('Permission denied')
    else:
        logger.error('Unexpected error: %d', e.errno)
```

在上面的例子中，变量e保存着异常OSError的实例。如果之后需要检查这个异常，比如要根据额外的状态码来进行处理，那这就很有用了。

需要注意的是，except子句是按照列出的顺序进行检查的，而第一个匹配成功的子句将得到执行。虽然这么做会有些病态，但是可以轻易地创建出多个except子句都可能匹配的情况。示例如下：

```
>>> f = open('missing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing
>>> try


:
...


f = open('missing')
... except OSError


:
...     print


('It failed')
... except


FileNotFoundError:
...     print


('File not found')
...



It failed
>>>
```

这里的except FileNotFoundError子句不会执行，因为OSError更加一般化，它也能匹配FileNotFoundError异常，而且它是首先列出的，因

此会先匹配执行。

给大家一个调试的小技巧，如果你不能完全确定某个特定异常的类层次结构，可以通过检查异常的**mro** 属性来快速查阅。示例如下：

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundError'>, <class 'OSError'>, <class 'Exceptio
<class 'BaseException'>, <class 'object'>)
>>>
```

以上列出的这些类中，只要排在BaseException之前的都可以用在except语句中。

14.7　捕获所有的异常

14.7.1　问题

我们想编写代码来捕获所有的异常。

14.7.2　解决方案

要捕获所有的异常，可以为Exception类编写一个异常处理程序，示例如下：

```
try

:
   ...
except Exception as

e:
   ...
   log('Reason:', e)     # Important!
```

除了SystemExit、KeyboardInterrupt和

GeneratorExit之外，上面的代码能够捕获所有的异常。如果也想捕获这些异常的话，只要把Exception修改为BaseException即可。

14.7.3　讨论

有时候当程序员没法记住某个复杂操作中可能产生的所有异常时，捕获所有的异常就成了他们唯一的支柱。同样，如果你不够小心的话，这也是写出无法调试的代码的绝佳方式。

正因为如此，如果选择捕获所有的异常，那么针对异常产生的实际原因做日志记录或报告就绝对是至关重要的了（例如，产生日志文件，或者将出错信息打印到屏幕上等）。如果不这么做，那么某个时刻你的大脑很可能会乱成一锅粥。考虑下面这个示例：

```
def

parse_int(s):
    try

:
        n = int(v)
    except Exception

:
```

```
        print


("Couldn't parse")
```

如果试着调用这个函数，它的行为是这样的：

```
>>> parse_int('n/a')
Couldn't parse
>>> parse_int('42')
Couldn't parse
>>>
```

此时，你可能会抓着脑袋想为什么它不能工作呢？现在假设函数被改写为如下形式：

```
def


parse_int(s):
    try


:
        n = int(v)
    except Exception as


e:
        print


("Couldn't parse")
        print
```

```
('Reason:', e)
```

在这种情况下，会得到如下形式的输出，能够清楚表明上述代码中出现了一个编程错误：

```
>>> parse_int('42')
Couldn't parse
Reason: global name 'v' is not defined
>>>
```

所有事情都是平等的，在处理异常的时候最好还是尽可能使用精确的异常类。但是，如果必须捕获所有的异常，那就要确保提供高质量的诊断信息，或者将异常传播出去，这样就不会丢失异常产生的原因。

14.8　创建自定义的异常

14.8.1　问题

我们正在构建一个应用，希望对底层的异常进行包装从而打造出自定义的异常类。这种自定义的异常在应用程序的上下文环境中可以包含更多的含义。

14.8.2　解决方案

创建新的异常是非常简单的——只要将它们定义成继承自Exception的类即可（也可以继承自其他已有的异常类型，如果这么做更有道理的话）。例如，如果正在编写网络编程相关的代码，则可能会像这样定义一些自定义的异常：

```
class NetworkError

(Exception

):
    pass
```

```
class HostnameError


(NetworkError):
    pass




class TimeoutError


(NetworkError):
    pass




class ProtocolError


(NetworkError):
    pass
```

用户能够以普通的方式来使用这些异常，示例如下：

```
try


:
    msg = s.recv()
except


TimeoutError as
```

```
e:
    ...
except


ProtocolError as


e:
...
```

14.8.3 讨论

自定义的异常类应该总是继承自内建的Exception类，或者继承自一些本地定义的基类，而这个基类本身又是继承自Exception的。虽然所有的异常也都继承自BaseException，但不应该将它作为基类来产生新的异常。BaseException是预留给系统退出异常的，比如KeyboardInterrupt或者SystemExit，以及其他应该通知应用程序退出的异常。因此，捕获这些异常并不适用于它们本来的用途。假设遵循这个约定，从BaseException继承而来的自定义异常将无法捕获，也不能通知应用程序关闭！

在自己的应用中使用自定义的异常，这使得任何需要阅读源代码的人能够更好地理解程序的行为。有一种设计上的考虑是通过继承机制将自定义

的异常归类到一起。在复杂的应用中，引入更高层的基类将不同的异常类归组到一起是很有意义的。这给了用户捕获细粒度错误的能力，比如：

```
try

:
    s.send(msg)
except

ProtocolError:
    ...
```

　　但同样也能够捕获粗粒度范围内的错误，比如：

```
try

:
    s.send(msg)
except

NetworkError:
    ...
```

如果打算定义一个新的异常并且改写Exception的**init** ()方法，请确保总是用所有传递过来的参数调用Exception.**init** ()。示例如下：

```
class CustomError


(Exception


):
    def


__init__(self, message, status):
        super().__init__(message, status)
        self.message = message
        self.status = status
```

这可能看起来有点古怪，但是Exception的默认行为就是接受所有传递过来的参数并将它们以元组的形式保存到.args属性中。Python中的其他组件以及各种各样的库都期望所有的异常都有一个.args属性，因此如果跳过了这一步，那么就会发现新创建的异常在特定上下文环境中表现出不正确的行为。为了说明对.args的使用，考虑下面的交互式会话，这里用到了内建的RuntimeError异常，注意在raise语句中可以使用多少个参数：

```
>>> try
```

```
:
...     raise RuntimeError

('It failed')
... except RuntimeError as

e:
...     print

(e.args)
...

('It failed',)
>>> try

:
...     raise RuntimeError

('It failed', 42, 'spam')
... except RuntimeError as

e:
...     print

(e.args)
...

('It failed', 42, 'spam')
>>>
```

要得到关于创建自定义异常的更多内容，请查阅Python文档（http://docs. python.org/3/tutorial/errors.html）。

14.9　通过引发异常来响应另一个异常

14.9.1　问题

我们想引发一个异常作为捕获另一个异常时的响应，但是希望在traceback回溯中同时包含这两个异常的有关信息。

14.9.2　解决方案

要将异常串联起来，可以用raise from语句来替代普通的raise。这么做能够提供这两个异常的有关信息。示例如下：

```
>>> def


example():
...     try


:
...


int('N/A')
...     except ValueError as


```

```
e:
...         raise RuntimeError


('A parsing error occurred') from e...




>>>




example()
Traceback (most recent call last):
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'

The above exception was the direct cause of the following except

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example
RuntimeError: A parsing error occurred
>>>
```

在traceback回溯中可以发现这两个异常都被捕获了。要捕获这样的异常，可以使用普通的except语句。但是，可以通过查看异常对象的**cause** 属性来跟踪所希望的异常链。示例如下：

```
try


:
    example()
except RuntimeError as
```

```
e:
    print


("It didn't work:", e)
if


e.__cause__:
    print


('Cause:', e.__cause__)
```

当在except语句块中引发另一个异常时，此时会产生异常链的隐式形式。示例如下：

```
>>> def


example2():
...     try


:
...


int('N/A')
...     except ValueError as


e:
...         print


("Couldn't parse:", err)
```

```
...



>>>
>>> example2()
Traceback (most recent call last):
  File "<stdin>", line 3, in example2
ValueError: invalid literal for int() with base 10: 'N/A'

During handling of the above exception, another exception occurre

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example2
NameError: global name 'err' is not defined
>>>
```

在这个例子中可以得到这两个异常的相关信息，但是这与第一个例子有所不同。在这种情况下，NameError异常是由于编程错误而产生的，并非是对解析错误的直接响应。因此在这种情况下，异常对象的**cause** 属性并没有被设置。相反，会把**context** 属性设置为前一个异常（即，ValueError）。

如果出于某种原因想阻止异常链的产生，可以使用raise from None来完成：

```
>>> def


example3():
```

```
...     try


:
...


int('N/A')
...     except ValueError


:
...         raise RuntimeError


('A parsing error occurred') from None...




...




>>> example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: A parsing error occurred
>>>
```

14.9.3 讨论

在设计代码的时候，对于在except块中使用raise语句的情况，大家应该特别小心。大部分情况下，这种raise语句都应该改为raise from。也就是说，我们应该采用下面这种风格：

```
try

:
    ...
except

SomeException as

e:
    raise

DifferentException() from e
```

这么做的原因在于我们需要显式将异常产生的原因串联起来。也就是说，Different Exception是直接响应SomeException。这两个异常间的关系会在traceback回溯中显式给出。

如果采用下面这种风格，还是可以得到异常链。但是通常这么做不能明确表达出异常链是程序员有意为之，还是由于无法预见的编程错误而产生的：

```
try
```

```
:
    ...
except


SomeException:
    raise


DifferentException()
```

当使用raise from语句时，就需明确表达出你希望引发第二个异常的意图。

最好不要像最后那个例子中那样抑制异常信息。尽管这么做产生的traceback回溯会比较短小，但同时也丢弃了对调试而言很有用的信息。任何事情没有绝对之分，通常最好还是尽可能多地保留调试信息为好。

14.10 重新抛出上一个异常

14.10.1 问题

我们在except块中捕获了一个异常，现在想将它重新抛出。

14.10.2 解决方案

只需要单独使用raise语句即可。示例如下：

```
>>> def


example():
...      try


:
...


int('N/A')
...     except ValueError


:
...               print


("Didn't work")
...               raise
```

```
...
>>> example()
Didn't work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

14.10.3　讨论

这个问题通常出现在需要对某个异常做出响应（比如记录日志、完成清理工作等），但之后希望将异常再传播出去时。一个非常常见的用途就是用在捕获所有异常的处理中：

```
try

:
    ...
except Exception as

e:
    # Process exception information in some way

    ...

    # Propagate the exception
```

```
    raise
```

14.11　发出告警信息

14.11.1　问题

我们想让自己的程序能够发出告警信息（例如，对废弃的功能或者使用上的问题进行告警提示）。

14.11.2　解决方案

要让程序产生告警信息，可以使用warnings.warn()函数。示例如下：

```
import warnings



def

func(x, y, logfile=None, debug=False):
    if

logfile is not

None:
        warnings.warn('logfile argument deprecated', Deprecation
```

```
)
    ...
```

warn()函数的参数是一条告警信息附带一个告警类别，通常类别为UserWarning、DeprecationWarning、SyntaxWarning、RuntimeWarning、ResourceWarning或者FutureWarning中的一种。

对告警信息的处理取决于如何执行解释器以及其他的相关配置。例如，如果用-W all选项来运行Python解释器，则会得到如下的输出：

```
bash % python3 -W all example.py
example.py:5: DeprecationWarning: logfile argument is deprecated
  warnings.warn('logfile argument is deprecated', DeprecationWarn
```

一般来说，告警信息只会发送到标准错误输出上。如果想把告警转换为异常，可以使用-W error选项：

```
bash % python3 -W error example.py
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    func(2, 3, logfile='log.txt')
  File "example.py", line 5, in func
    warnings.warn('logfile argument is deprecated', DeprecationWa
```

```
DeprecationWarning: logfile argument is deprecated
bash %
```

14.11.3 讨论

为了维护软件以及帮助用户更好地使用软件，产生告警信息常常是一项有用的技术。这么做可以让那些没必要上升到异常层面的问题以告警信息的形式表达出来。比如说，如果打算修改某个库或者框架的行为，可以针对打算修改的部分启用告警信息，同时在一段时间内仍然提供向后兼容性。也可以提醒用户有关用法方面的问题。

在内建的warning库中还有另一个关于告警应用的示例。下面的例子用来说明当未关闭文件对象就打算将其销毁时所产生的告警信息：

```
>>> import warnings



>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del


f
__main__:1: ResourceWarning: unclosed file <_io.TextIOWrapper nar
 mode='r' encoding='UTF-8'>
>>>
```

默认情况下，并非所有的告警信息都会显示出来。-W选项能够控制告警信息的输出。-W all将会输出所有的告警信息，而-W ignore选项会忽略所有的告警，-W error会将告警转换为异常。另外一种替代方案就是使用warnings.simplefilter()函数来控制输出，上面的示例采用的正是这个方法。参数“always”使得所有的告警信息都会显示，而“ignore”则表示忽略所有的告警，“error”则会把告警转换为异常。

对于简单的情况，这就是所有需要了解的有关产生告警信息的知识。warnings模块提供了许多与信息过滤以及处理告警信息相关的高级配置选项。更多信息可参阅Python文档（http://docs.python.org/3/library/warnings.html ）。

14.12 对基本的程序崩溃问题进行调试

14.12.1 问题

我们的程序崩溃了，我们希望通过一些简单的策略来调试它。

14.12.2 解决方案

如果程序由于产生异常而崩溃了，可以通过python3 -i someprogram.py的方式来运行程序。这么做可以简单地查看产生问题的原因。一旦程序终止，-i选项就会开启一个交互式shell，这里就可以对程序运行的环境做一番探究了。例如，假设有如下的代码：

```
# sample.py



def


func(n):
    return
```

```
n + 10

func('Hello')
```

通过python3 -i来运行程序会产生下列输出：

```
bash % python3 -i sample.py
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    func('Hello')
  File "sample.py", line 4, in func
    return


n + 10
TypeError: Can't convert 'int' object to str implicitly
>>> func(10)
20
>>>
```

如果这么做还看不出什么明显的问题，那么在程序崩溃之后还可以加载Python调试器来助阵。示例如下：

```
>>> import pdb



>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
```

```
sample.py(6)<module>()
-> func('Hello')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Hello'
(Pdb) q
>>>
```

如果我们的代码深埋在一个难以获取交互式shell的环境中（比如在服务器中），通常可以捕获错误并自己生成traceback回溯。示例如下：

```
import traceback



import sys




try


:
    func(arg)
except


:
    print


('**** AN ERROR OCCURRED ****')
    traceback.print_exc(file=sys.stderr)
```

如果程序并没有崩溃只是产生错误的结果，又或者我们只是想了解程序究竟是如何工作的，那么在代码中感兴趣的位置上插入一些print()调用是完全合理的。但是，如果真的打算这么做，这里有一些我们可能会感兴趣的相关技术。首先，traceback.print_stack()函数会在程序调用它的地方立刻打印出调用栈的信息。示例如下：

```
>>> def


sample(n):
...     if


n > 0:
...


sample(n-1)
...     else


:
...


traceback.print_stack(file=sys.stderr)
...



>>> sample(5)
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in sample
```

```
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 5, in sample
>>>
```

作为替代方案，也可以在程序的任意位置通过调用pdb.set_trace()来手动加载调试器：

```
import pdb



def


func(arg):
    ...
    pdb.set_trace()
    ...
```

这对于研究大型程序的内部原理、了解程序的控制流或者函数参数都是非常有用的技术。例如，一旦启动了调试器，就可以通过print来观察变量，或者输入命令比如w来获取栈回溯信息。

14.12.3 讨论

不要把调试弄的过于复杂。简单的错误常常可以通过阅读程序的traceback回溯来解决（实际错误通常是traceback的最后一行）。如果你正处于开发过程中，而你只是想获得一些诊断信息，那么在代码中插入一些print()函数也能很好的完成任务（只是稍后要记得将这些打印语句去掉）。

调试器的常见用途就是对已经崩溃的函数中的变量进行检查。知道如何在程序崩溃之后进入调试器是一项有用的技能。

如果要研究一个特别复杂的程序，其底层的控制流并不明显，那么插入像pdb.set_trace()这样的语句也是十分有帮助的。从本质上说，程序会一直运行，直到遇到set_trace()调用为止，此时会立刻进入调试器。这之后就可以好好利用调试器的功能了。

如果使用IDE来做Python开发，一般来说IDE都会提供自己的调试接口。调试接口要么是构建在pdb之上的，要么就取代了pdb。可以参考你的IDE手册以获得更多的信息。

14.13 对程序做性能分析以及计时统计

14.13.1 问题

我们想知道程序在运行时把时间都花在了哪些地方，并且想对运行时间做计时统计。

14.13.2 解决方案

如果只是想简单地对整个程序做计时统计，通常使用UNIX下的time命令就足够了。示例如下：

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys   0m0.098s
bash %
```

再来看看另一个极端。如果想针对程序的行为产生一份详细的报告，那么可以使用cProfile模块：

```
bash % python3 -m cProfile someprogram.py
         859647 function calls in 16.016 CPU seconds

   Ordered by: standard name
```

```
   ncalls tottime  percall  cumtime  percall filename:lineno(functio
   263169 0.080   0.000   0.080   0.000   someprogram.py:16(frang
      513 0.001   0.000   0.002   0.000   someprogram.py:30(gener
   262656 0.194   0.000   15.295  0.000   someprogram.py:32(<gene
        1  0.036   0.036   16.077  16.077  someprogram.py:4(<modu
   262144 15.021  0.000   15.021  0.000   someprogram.py:4(in_man
        1 0.000   0.000   0.000   0.000   os.py:746(urandom)
        1 0.000   0.000   0.000   0.000   png.py:1056(_readable)
        1 0.000   0.000   0.000   0.000   png.py:1073(Reader)
        1 0.227   0.227   0.438   0.438   png.py:163(<module>)
      512 0.010   0.000   0.010   0.000   png.py:200(group)
   ...
bash %
```

对代码做性能分析，更常见的情况则处于上述两个极端情况之间。比如，我们可能已经知道了代码把大部分运行时间都花在某几个函数上了。要对函数进行性能分析，使用装饰器就能办到。示例如下：

```
# timethis.py

import time

from functools import

wraps

def
```

```
timethis(func):
    @wraps(func)
    def


wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print


('{}.{} : {}'.format(func.__module__, func.__name__, end - start
        return


r
return


wrapper
```

要使用这个装饰器，只要简单地将其放在函数定义之前，就能得到对应函数的计时信息了。示例如下：

```
>>> @timethis
... def


countdown(n):
...     while


n > 0:
...
```

```
n -= 1
...



>>> countdown(10000000)
__main__.countdown : 0.803001880645752
>>>
```

要对语句块进行计时统计，可以定义一个上下文管理器来实现。示例如下：

```
from contextlib import


contextmanager

@contextmanager
def


timeblock(label):
    start = time.perf_counter()
    try


:
        yield



    finally


:
        end = time.perf_counter()
```

```
        print

('{} : {}'.format(label, end - start))
```

下面的例子演示了这个上下文管理器是如何工作的：

```
>>> with

timeblock('counting'):
...

n = 10000000
...     while

n > 0:
...

n -= 1
...


counting : 1.5551159381866455
>>>
```

如果要对短小的代码片段做性能统计，timeit模块会很有帮助。示例如下：

```
>>> from timeit import


timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

timeit会执行第一个参数中指定的语句一百万次，然后计算时间。第二个参数是一个配置字符串，在运行测试之前会先执行以设定好环境。如果需要修改迭代的次数，只需要提供一个number参数即可。示例如下：

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

14.13.3　讨论

请注意，在进行性能统计时，任何得到的结果都是近似值。解决方案中使用的函数time.perf_counter()能够提供给定平台上精度最高的计时器。但是，它计算的仍然是墙上时间（wall-

clock time），这会受到许多不同因素的影响，例如机器当前的负载。

如果相对于墙上时间，我们更感兴趣的是进程时间，那么可以使用time.process_time()来替代。示例如下：

```
from functools import


wraps
def


timethis(func):
    @wraps(func)
    def


wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print


('{}.{} : {}'.format(func.__module__, func.__name__, end - start
        return


r
    return


wrapper
```

最后但同样重要的是，如果打算进行详细的计时统计分析，请确保先阅读time、timeit以及其他相关模块的文档。这样才能理解不同系统平台之间的重要差异以及其他一些缺陷。

本书13.13节也介绍了一个相关的主题，即创建一个秒表定时器。

14.14 让你的程序运行得更快

14.14.1 问题

我们的程序运行得太慢了，我们想让它提速，但不使用那些极端的解决方案，比如C扩展或JIT编译器。

14.14.2 解决方案

尽管关于优化的第一原则也许是“不优化”，但第二原则几乎肯定是“不要优化那些不重要的部分”。基于这两个原则，如果我们的程序运行的很慢，应该采用14.13节中讨论的方法开始对代码进行性能分析。

多半时候我们都会发现程序把大量的时间花在了几个“热点”（hotspot）上，比如处理数据时的内层循环。一旦确认了这些“热点”，就可以使用以下各小节中介绍的技术让程序运行得更快。

使用函数

有很多程序员开始使用Python时都用它来编写

一些简单的脚本。当编写脚本时，很容易陷入只管编写代码而不重视程序结构的怪圈。例如：

```
# somescript.py

import sys

import csv

with

open(sys.argv[1]) as

f:
    for

row in

csv.reader(f):
    # Some kind of processing

    ...
```

一个鲜为人知的事实是，像上面这样定义在全局范围内的代码运行起来比定义在函数中的代码要慢。速度的差异与局部变量和全局变量的实现机制有关（涉及局部变量的操作要更快）。因此，如果想让程序运行得更快，只要将脚本中的语句放入一个函数中即可：

```
# somescript.py

import sys

import csv

def

main(filename):
    with

open(filename) as

f:
    for

row in

csv.reader(f):
        # Some kind of processing
```

```
        ...
main(sys.argv[1])
```

运行速度的差异与所执行的处理有很大关系，但根据我们的经验，提升15%～30%的情况并非罕见。

有选择性的消除属性访问

每次使用句点操作符（.）来访问属性时都会带来开销。在底层，这会触发调用特殊方法，比如**getattribute** ()和**getattr** ()，而调用这些方法常常会导致做字典查询操作。

通常可以通过from module import name的导入形式以及选择性地使用绑定方法（bound method）来避免出现属性查询操作。为了说明清楚，考虑下面的代码片段：

```
import math



def
```

```
compute_roots(nums):
    result = []
    for


n in


nums:
        result.append(math.sqrt(n))
    return


result

# Test



nums = range(1000000)
for


n in


range(100):
    r = compute_roots(nums)
```

当在我们的机器上测试时，这个程序运行了大约40秒。现在将compute_roots()函数修改为如下形式：

```
from math import
```

```
sqrt

def


compute_roots(nums):
result = []
result_append = result.append
for


n in


nums:
    result_append(sqrt(n))
return


result
```

这个版本的运行时间大约是29秒。两个版本间的唯一区别就在属性访问上，第二个版本消除了对属性的访问。与其使用math.sqrt()，现在代码可直接使用sqrt()。此外，result.append()方法现在被放置在一个局部变量result_append中，然后再在内层循环中重复使用它。

但是，必须要强调的是，只有在频繁执行的代码中做这些修改才有意义，比如在循环中。因此，这种优化技术适用的场景需要经过仔细挑选。

理解变量所处的位置

前面已经说过了，访问局部变量比全局变量要更快。对于需要频繁访问的名称，想提高运行速度，可以通过让这些名称尽可能成为局部变量来达成。例如，考虑下面这个修改过的compute_roots()函数：

```
import math



def


compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for


n in


nums:
        result_append(sqrt(n))
    return


result
```

在这个版本中，sqrt方法已经从math模块中提

取出来并放置在一个局部变量中。如果运行这份代码，现在的运行时间大约是25秒（比上一个版本的29秒又有所提升）。这次提升就是因为查找局部变量sqrt比在全局范围内查找sqrt要更快。

当使用类的时候，局部参数同样能起到提速的效果。一般来说，查找像self.name这样的值会比访问一个局部变量要慢很多。在内层循环中将需要经常访问的属性移到局部变量中来会很划算。示例如下：

```
# Slower

class SomeClass

:
    ...
    def

method(self):
         for

x in

s:
             op(self.value)

# Faster
```

```
class SomeClass


:
    ...
    def


method(self):
         value = self.value
         for


x in


s:
             op(value)
```

避免不必要的抽象

任何时候当使用额外的处理层比如装饰器（decorator）、属性（property）或者描述符（descriptor）来包装代码时，代码的运行速度就会变慢。作为示例，参考下面这个类：

```
class A


:
    def


__init__(self, x, y):
```

```
        self.x = x
        self.y = y
    @property
    def


y(self):
        return


self._y
    @y.setter
    def


y(self, value):
        self._y = value
```

现在做一个简单的计时测试：

```
>>> from timeit import


timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

可以看到，访问property属性y比访问普通的属性x慢了不止一点，而是大约慢了4.5倍。如果这种差异对你而言很重要，你应该问问自己是否真的有

必要将y定义为property属性。如果不是，那么去掉property重新用普通的属性来替代即可。不能仅仅因为在其他的编程语言中使用getter/setter函数非常普遍，就错误地把这种编程风格应用到Python上来。

使用内建的容器

内建的数据类型比如字符串、元组、列表、集合以及字典都是用C语言实现的，速度非常快。如果倾向于构建自己的数据结构作为替代（例如，链表、平衡二叉树等），想在速度上和内建的数据结构相抗衡即使并非不可能也绝对会相当困难。因此，通常最好还是直接使用内建的数据结构。

避免产生不必要的数据结构或者拷贝动作

有时候程序员会在不必要的情况下忘乎所以地创建一些不必要的数据结构。例如，有的人可能会编写出如下的代码：

```
values = [x for


x in


sequence]
squares = [x*x for


```

```
x in

values]
```

也许这里的想法是先将一些值收集到一个列表中，然后再对列表进行操作，比如列表推导。但是，这里的第一个列表完全是没有必要的。只要把代码写成这样 即可：

```
squares = [x*x for

x in

sequence]
```

与此相关的是，那些对Python中共享值的行为过于偏执的程序员，他们编写的代码需要好好检查一番。过度使用像copy.deepcopy()这样的函数就是一个信号，这表示代码的编写者不完全理解或者说信赖Python的内存模型。在这样的代码中消除那些不必要的拷贝应该是安全的。

14.14.3 讨论

在进行优化之前，首先分析一下正在使用的算法通常都是很值得的。把算法的复杂度切换为O(nlgn)所带来的性能提升绝对比费力调整一个O(n**2)的实现要高　得多。

如果仍然决定必须优化，那么就从大的方向考虑。一般来说，我们不会针对程序的每个部分都去优化，因为这样的修改会使得代码难以阅读和理解。相反，只针对已知的性能瓶颈做修改，比如内层循环。

我们需要特别留意微优化（micro-optimization）所带来的结果。比如，考虑下面这两种创建字典的方法：

```
a = {
    'name' : 'AAPL',
    'shares' : 100,
    'price' : 534.22
}

b = dict(name='AAPL', shares=100, price=534.22)
```

后一种方法的好处在于不需要输入那么多字符（不需要将键名括起来）。但是，如果对上述两个代码片段做一个性能测试对比，就会发现使用dict()的版本要慢3倍！有了这种认识之后，我们可能会倾向于将自己的代码扫描一遍，把每个用到dict()的

地方都用更加冗长的方法替换掉。但是，聪明的程序员只会把精力集中在程序中实际会产生性能影响的地方，比如内层循环。而其他地方的速度差异根本就是无关紧要的。

另一方面，如果我们对性能提升的要求已经远远超出了本节所讨论的这几种简单技术，那就需要考虑使用基于即时编译（just-in-time compilation）技术的工具了。例如，PyPy项目（http://pypy.org ）就是对Python解释器的重新实现，可以分析你的程序并针对频繁执行的部分生成原始机器码。有时候能使Python程序的运行速度快上一个数量级，常常能接近（甚至超越）C代码的执行速度。不幸的是，在写作本书时PyPy还没能完全支持Python 3。因此，这是未来需要关注的问题。此外也可以考虑Numba项目（http://numba.pydata.org ）。Numba是一个动态编译器，我们可以选择需要优化的Python函数，然后用装饰器来装饰。这些函数就会通过LLVM（http://llvm.org ）编译成原始的机器码。这种方法同样能够获得显著的性能提升。但是和PyPy一样，Numba对Python 3的支持应该还只能看做是试验阶段。

最后但同样重要的是，请牢记John Ousterhout（Tcl/Tk语言发明者）的名言：最好的性能提升就是从不能工作转变为可以工作。（The

best performance improvement is the transition from the nonworking to the working state.）在确实需要优化之前别担心这个问题。确保让程序能够正常工作总是比让它运行的更快要更加重要（至少在最初阶段是如此）。

第15章　C语言扩展

本章将讨论从Python中访问C代码的问题。许多Python的内建库都是用C语言编写的，能够访问C代码对于让Python同现有的库进行交互是十分重要的一环。此外，如果我们面临着将扩展代码从Python 2移植到Python 3中，那么这也是需要重点学习的部分。

尽管Python提供了广泛的C语言编程API，但实际上有着多种不同的方法来应对C代码。与其针对每个可能的工具和技术都给出详尽的参考，我们采用的方法是把重点放在小段的C代码上，用有代表性的示例来展示如何同C代码交互。目的是提供一系列的编程模板，有经验的程序员可以展开后供自己使用。

以下就是我们在后续大部分章节中需要打交道的C代码：

```
/* sample.c */


_method
#include <math.h>

/* Compute the greatest common divisor */
```

```
int


gcd(int


x, int


y) {
    int


g = y;
    while


(x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return


g;
}

/* Test if (x0,y0) is in the Mandelbrot set or not */



int


in_mandel(double


x0, double


y0, int
```

```
n) {
  double


x=0,y=0,xtemp;
  while


(n > 0) {
    xtemp = x*x - y*y + x0;
    y = 2*x*y + y0;
    x = xtemp;
    n -= 1;
    if


(x*x + y*y > 4) return


0;
  }
  return


1;
}

/* Divide two numbers */



int


divide(int


a, int


b, int


*remainder) {
```

```
  int


quot = a / b;
  *remainder = a % b;
  return


quot;
}

/* Average values in an array */



double


avg(double


*a, int


n) {
  int


i;
  double


total = 0.0;
  for


(i = 0; i < n; i++) {
    total += a[i];
  }
  return


total / n;
}
```

```
/* A C data structure */



typedef struct


Point {
    double


x,y;
} Point;

/* Function involving a C data structure */



double


distance(Point *p1, Point *p2) {
   return


hypot(p1->x - p2->x, p1->y - p2->y);
}
```

这份代码包含了大量C编程中用到的不同特性。首先，有一些简单的函数如gcd()和is_mandel()。而divide()则是C函数中返回多个值的一个例子，其中一个值是通过指针参数返回的。avg()函数遍历了C数组并做了数据转换。Point和distance()函数涉及了C结构体。

后面所有的小节都假设前面这些C代码保存在

名为*sample.c* 的文件中，声明可以在*sample.h* 中找到，而且代码已经被编译为libsample库，可以将其链接到其他的C代码中。编译和链接的具体细节在不同的系统之间有所区别，但这不是我们主要关注的问题。我们假设如果你正在同C代码打交道，则你已经了解了这些知识。

15.1　利用ctypes来访问C代码

15.1.1　问题

我们有一些C函数已经被编译为共享库或者DLL了。我们想从纯Python代码中直接调用这些函数，而不必额外编写C代码或者使用第三方的扩展工具。

15.1.2　解决方案

对于用C语言编写的小程序，使用Python标准库中的ctypes模块来访问通常是非常容易的。要使用ctypes，必须首先确保想要访问的C代码已经被编译为与Python解释器相兼容（即，采用同样的体系结构、字长、编译器等）的共享库了。对于本小节来说，假设已经创建了共享库libsample.so，其中包含了本章介绍中所示的那些代码。我们进一步假设文件libsample.so与接下来展示的sample.py放置在同一个目录中了。

要访问这个共享库，需要创建一个Python模块来包装它，示例如下：

```
# sample.py

import ctypes

import os

# Try to locate the .so file in the same directory as this file

_file = 'libsample.so'
_path = os.path.join(*(os.path.split(__file__)[:-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)

gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)

in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)

_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(c
_divide.restype = ctypes.c_int
```

```
def


divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)
    return


quot,rem.value

# void avg(double *, int n)



# Define a special type for the 'double *' argument



class DoubleArrayType


:
    def


from_param(self, param):
        typename = type(param).__name__
        if


hasattr(self, 'from_' + typename):
            return


getattr(self, 'from_' + typename)(param)
        elif


isinstance(param, ctypes.Array):
            return


param
```

```
        else

:
            raise TypeError

("Can't convert %s" % typename)

    # Cast from array.array objects


    def

from_array(self, param):
        if

param.typecode != 'd':
            raise TypeError

('must be an array of doubles')
        ptr, _ = param.buffer_info()
        return

ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))

    # Cast from lists/tuples


    def

from_list(self, param):
        val = ((ctypes.c_double)*len(param))(*param)
        return

val
```

```
    from_tuple = from_list

    # Cast from a numpy array



    def


from_ndarray(self, param):
        return


param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()
_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double

def


avg(values):
    return


_avg(values, len(values))

# struct Point { }



class Point


(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

# double distance(Point *, Point *)



distance = _mod.distance
```

```
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point
distance.restype = ctypes.c_double
```

如果一切顺利，现在应该可以加载这个模块并使用相应的C函数了。例如：

```
>>> import sample



>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

15.1.3　讨论

本节中有几个地方需要进行讨论。第一个问题是关于将C和Python代码打包在一起。如果要用ctypes来访问自己编译的C代码，得确保把共享库放

在sample.py模块可以找得到的地方。一种可能是将得到的.so文件与所支撑的Python代码放在同一个目录中。这正是本节给出的解决方案中首先完成的——*sample.py* 查询__file__变量，看看自己被安装到何处，然后在同样的目录下构建一个路径指向*libsample.so* 文件。

如果要把C库安装到别处，那么必须相应地调整路径。如果C库已经作为标准库安装到你的机器上了，那么可以使用ctypes.util.find_library()函数。示例如下：

```
>>> from ctypes.util import


find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>
```

再次申明，如果ctypes无法找到C库则不能继续工作。因此，需要花几分钟时间考虑一下该如何安装库。

一旦知道了C库的位置，可以使用

ctypes.cdll.LoadLibrary()来加载。在解决方案中，_path是指向共享库的完整路径，而下列语句则用来加载C库：

```
_mod = ctypes.cdll.LoadLibrary(_path)
```

一旦成功加载了C库，我们需要编写代码来提取特定的符号并为其附上类型签名。这正是由如下代码完成的：

```
# int in_mandel(double, double, int)



in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_
in_mandel.restype = ctypes.c_int
```

在这份代码中，.argtypes属性是一个元组，其中包含了函数的输入参数，而.restype表示返回类型。ctypes中定义了许多类型对象（例如c_double、c_int、c_short、c_float等），它们用来代表常见的C数据类型。如果想要Python传递正确的参数类型并对数据做正确的转换，那么给值附上类型签名就是至关重要的了（如果不这么做，不仅代码不会正

常工作，而且还会使得整个解释器进程崩溃）。

使用ctypes时，一个多少有些棘手的地方在于原始的C代码中可能会用到一些惯用法，而它们在概念上不能清晰地映射到Python中。divide()函数就是个很好的例子，因为它是通过其中一个参数来返回值的。尽管这在C中是非常常见的技术，但放在Python中往往就不清楚应该如何工作了。例如，我们不能直接像这样做：

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected 
instance instead of int
>>>
```

就算这么做行的通，也会违反Python中整数是不可变对象的事实，可能会导致整个解释器进程卡死在黑洞中。对于涉及指针的参数，通常必须构建一个兼容的ctypes对象，然后像下面这样传入：

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

这里我们创建了一个ctypes.c_int对象，并把它作为指针对象传递给函数。与普通的Python整数不同，c_int对象是可变的。可以根据需要通过.value属性来获取或修改值。

对于那些C调用约定（calling convention）属于非Pythonic（Pythonic是俚语，表示按照Python的方式来优雅的工作）的情况，通常都需要编写一个小型的包装函数来处理。在解决方案中，这个包装函数使得divide()函数用一个元组来返回两个结果值：

```
# int divide(int, int, int *)



_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ct
_divide.restype = ctypes.c_int

def


divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return


quot, rem.value
```

avg()函数则带来了全新的挑战。底层的C代码期望接收一个指针以及长度来代表一个数组。但是从Python的角度来看，我们必须考虑下列问题：什么是数组？它是列表还是元组？亦或是array模块中的array对象？是numpy中的数组吗？还是以上都有可能呢？在实践中，一个Python“数组”可能有着许多不同的形式，而且也许我们会想支持这多种可能。

类DoubleArrayType展示了如何处理这种情况。在这个类中我们定义了方法from_param()。这个方法的任务就是接受一个单独的参数并将其范围缩小为一个兼容的ctypes对象（在本例中就是指向ctypes.c_double的指针）。在from_param()中，我们可以自由地做任何想做的事。在我们的解决方案中，参数的类型名被提取出来并发送给更加具体的方法。例如，如果传递的是列表，则类型名就是list，调用的就是from_list()方法。

对于列表和元组，from_list()方法会执行转换到ctypes数组对象的操作。这看起来有点古怪，但下面是将列表转换为ctypes数组的交互式例子：

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
```

```
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

对于array对象，from_array()方法会提取底层的内存指针并将其转换为一个ctypes指针对象。示例如下：

```
>>> import array



>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>
```

from_ndarray()则对numpy数组做了处理。

通过定义DoubleArrayType类并在avg()的类型签名中使用，可以看到，函数可接受多种不同形式的数组输入：

```
>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
>>>
```

本节的最后部分是展示如何同简单的C结构体打交道。对于结构体来说，只用定义一个类，并在其中包含适当的字段和类型，示例如下：

```
class Point
(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

一旦定义完成，就可以在类型签名中使用它，也可以在需要实例化结构体对象的代码中使用。示例如下：

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

最后再多说几句：如果所有你想做的只是在Python中访问几个C函数，那么ctypes是很有用的库。但是，如果打算访问一个庞大的C库，就应该看看其他的方法，比如Swig（见15.9节）或者Cython（见15.10节）。

大型库的主要问题在于由于ctypes并不是全自动化处理的，我们将不得不花费大量时间来编写所有的类型签名，就像示例中的那样。根据库的复杂程度，我们可能也不得不编写大量的小型包装函数和支撑类（类似于DoubleArrayType）。此外，除非完全理解了所有C接口的底层细节，包括内存管理和错误处理，否则很容易会让Python由于段错误、非法访问或其他类似的错误而产生灾难性的崩

溃。

作为ctypes之外的选择，读者可以去看看CFFI（http://cffi.readthedocs.org/en/latest ）。CFFI提供了很多相同的功能，但使用的是C的语法，而且支持更多高级的C代码。在写作本节时，相对来说CFFI依然是一个很新的项目，但对它的使用已经得到了极大的增长。甚至有一些关于在今后的Python版本中将其纳入到Python标准库中的讨论。因此，CFFI绝对是值得去留意的项目。

15.2　编写简单的C语言扩展模块

15.2.1　问题

我们想不依赖任何其他工具直接用Python的扩展API编写一个简单的C语言扩展模块。

15.2.2　解决方案

对于简单的C代码，手工创建一个扩展模块是很简单直接的。作为第一步，可能要确保自己的C代码有一个合适的头文件。比如：

```
/* sample.h */



#include <math.h>

extern int


gcd(int


, int


);
extern int
```

```
in_mandel(double

x0, double

y0, int

n);
extern int

divide(int

a, int

b, int

*remainder);
extern double

avg(double

*a, int

n);

typedef struct

Point {
      double

x,y;
} Point;
```

```
extern double

distance(Point *p1, Point *p2);
```

通常情况下这个头文件会对应于一个单独编译好的库。带着这个假设，下面是一个C语言扩展模块的样例，用来说明编写扩展函数的基础：

```
#include "Python.h"
#include "sample.h"

/* int gcd(int, int) */



static


PyObject *py_gcd(PyObject *self, PyObject *args) {
  int


x, y, result;

  if


(!PyArg_ParseTuple(args,"ii", &x, &y)) {
    return


NULL;
  }
  result = gcd(x,y);
  return
```

```
Py_BuildValue("i", result);
}

/* int in_mandel(double, double, int) */



static


PyObject *py_in_mandel(PyObject *self, PyObject *args) {
  double


x0, y0;
  int


n;
  int


result;

  if


(!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
    return


NULL;
  }
  result = in_mandel(x0,y0,n);
  return


Py_BuildValue("i", result);
}

/* int divide(int, int, int *) */
```

```
static


PyObject *py_divide(PyObject *self, PyObject *args) {
  int


a, b, quotient, remainder;
  if


(!PyArg_ParseTuple(args, "ii", &a, &b)) {
    return


NULL;
  }
  quotient = divide(a,b, &remainder);
  return


Py_BuildValue("(ii)", quotient, remainder);
}

/* Module method table */



static


PyMethodDef SampleMethods[] = {
  {"gcd", py_gcd, METH_VARARGS, "Greatest common divisor"},
  {"in_mandel", py_in_mandel, METH_VARARGS, "Mandelbrot test"},
  {"divide", py_divide, METH_VARARGS, "Integer division"},
  { NULL, NULL, 0, NULL}
};

/* Module structure */



static struct
```

```
PyModuleDef samplemodule = {
  PyModuleDef_HEAD_INIT,
  "sample",          /* name of module */



  "A sample module", /* Doc string (may be NULL) */



  -1,                  /* Size of per-interpreter state or -1 */



  SampleMethods       /* Method table */



};

/* Module initialization function */



PyMODINIT_FUNC
PyInit_sample(void


) {
  return


PyModule_Create(&samplemodule);
}
```

为了构建扩展模块，需要创建一个*setup.py* 文件，看起来是这样的：

```
# setup.py

from distutils.core import

setup, Extension

setup(name='sample',
     ext_modules=[
       Extension('sample',
                 ['pysample.c'],
                 include_dirs = ['/some/dir'],
                 define_macros = [('FOO','1')],
                 undef_macros = ['BAR'],
                 library_dirs = ['/usr/local/lib'],
                 libraries = ['sample']
                 )
       ]
)
```

现在，要构建出目标库，只需要用python3 buildlib.py build_ext --inplace。示例如下：

```
bash % python3 setup.py build_ext --inplace
running build_ext
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-p
 -I/usr/local/include/python3.3m -c pysample.c
 -o build/temp.macosx-10.6-x86_64-3.3/pysample.o
gcc -bundle -undefined dynamic_lookup
build/temp.macosx-10.6-x86_64-3.3/pysample.o \

 -L/usr/local/lib -lsample -o sample.so
bash %
```

如上所示，这样就创建了一个名为sample.so的共享库。编译结束后，应该就可以开始将其当做一个Python模块来导入了：

```
>>> import sample



>>> sample.gcd(35, 42)
7
>>> sample.in_mandel(0, 0, 500)
1
>>> sample.in_mandel(2.0, 1.0, 500)
0
>>> sample.divide(42, 8)
(5, 2)
>>>
```

如果打算在Windows上尝试这些步骤，可能需要花点时间设置构建环境，以便正确生成扩展模块。Python的二进制发行版通常是用微软的Visual Studio来构建的。要让扩展模块正常工作，我们可能也要用相同或兼容的工具来编译。具体请参见Python的有关文档（http://docs.python.org/3/extending/windows.html）。

15.2.3　讨论

在尝试手工编写任何类型的扩展前，查阅Python文档中的“扩展和内嵌Python解释器”（Extending and Embedding the Python Interpreter）一节是至关重要的。Python的C语言扩展API很庞大，在这里重复所有的API是不现实的。但是，我们可以就最重要的部分在此讨论。

首先，在扩展模块中编写的函数通常都有着如下的共同原型：

```
static
PyObject *py_func(PyObject *self, PyObject *args) {
  ...
}
```

PyObject是一个C数据类型，表示任意的Python对象。从很高的层次来看，一个扩展函数就是一个C函数，它接受一组Python对象（在PyObject *args中）并返回一个新的Python对象作为结果。对于简单的扩展函数来说，函数的self参数是用不到的，但是当想在C中定义新的类或对象类型时就会派上用场了（例如，如果扩展函数是类的一个方法，那

么self就会用来表示对象实例）。

PyArg_ParseTuple()函数用来将值从Python转换为C语言中的表示。作为输入，它接受一个格式化字符串用来表示所需的值类型，例如“i”表示整数，而“d”表示double型浮点数。此外，它还接受C变量的地址作为参数，用来放置转换后的结果。PyArg_ParseTuple()会对参数的数量和类型做许多检查。如果在格式化字符串中发现有任何不匹配的项，则会产生一个异常并返回NULL。通过对参数的检查以及返回NULL，在调用端就会产生适当的异常了。

函数Py_BuildValue()用来从C数据类型创建出对应的Python对象。它也接受一个格式化代码用来表示所需的类型。在扩展函数中，它用来将结果返回给Python。Py_BuildValue()的一个特性是它可以构建类型更加复杂的对象，比如元组和字典。在针对py_divide()的代码中，我们已经展示了一个返回元组的例子。但是，下面还有一些更多的示例：

```
return

Py_BuildValue("i", 34);         // Return an integer
return

Py_BuildValue("d", 3.4);         // Return a double
```

```
return

Py_BuildValue("s", "Hello");     // Null-terminated UTF-8 string
return

Py_BuildValue("(ii)", 3, 4);     // Tuple (3, 4)
```

在任何扩展模块代码的底部，我们都会找到一个像示例中的SampleMethods这样的函数表。这张表列出了C函数、在Python中所用的名称以及文档字符串。所有的模块都需要指定一个这样的表，它会在模块初始化时用到。

最后，函数PyInit_sample()是模块的初始化函数，当模块首次导入时会调用执行。它的主要工作就是把模块对象注册到解释器中。

作为最后的说明，必须要强调的是，关于用C函数来扩展Python，还有相当多的内容没有在这里给出（事实上，Python的C API中包含了超过500个函数）。你应该把本节当做入门的踏脚石。要完成更多功能，可以从函数PyArg_ParseTuple()和Py_BuildValue()的文档开始，然后从那儿开始扩展。

15.3　编写一个可操作数组的扩展函数

15.3.1　问题

我们想编写一个C扩展函数来操作数组型数据，数组可能会通过array模块或NumPy这样的库来创建。但是，我们想让自己的函数变得通用，而不必具体于任何一个创建数组的库。

15.3.2　解决方案

要以可移植的方式来接收和处理数组，应该编写使用了Buffer Protocol（http://docs.python.org/3/c-api/buffer.html）的代码。下面是一个手写的C扩展函数示例，它接受数组数据并调用本章介绍部分给出的avg(double *buf, int len)函数：

```
/* Call double avg(double *, int) */

static
```

```
PyObject *py_avg(PyObject *self, PyObject *args) {
  PyObject *bufobj;
  Py_buffer view;
  double

result;
  /* Get the passed Python object */

  if

(!PyArg_ParseTuple(args, "O", &bufobj)) {
    return

NULL;
  }

  /* Attempt to extract buffer information from it */

  if

(PyObject_GetBuffer(bufobj, &view,
      PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    return

NULL;
  }
  if

(view.ndim != 1) {
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional a
    PyBuffer_Release(&view);
    return

NULL;
```

```
  }

  /* Check the type of items in the array */



  if


(strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of double
    PyBuffer_Release(&view);
    return


NULL;
  }

  /* Pass the raw buffer and size to the C function */



  result = avg(view.buf, view.shape[0]);

  /* Indicate we're done working with the buffer */



  PyBuffer_Release(&view);
  return


Py_BuildValue("d", result);
}
```

下面的示例展示了这个扩展函数是如何工作的：

```
>>> import array
```

```
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])
2.0
>>>
```

15.3.3 讨论

将数组对象传递给C函数可能是在编写扩展函数中最常遇到的情况之一了。从图像处理到科学计算领域，有大量的Python应用程序都依赖于对数组

的高效处理。通过编写可以接受并操作数组的代码，就可以编写自定义的代码来很好地应用到这些应用之上，而不是鼓捣出某种自定义的解决方案只能用在自己的代码中。

示例代码的核心就在PyBuffer_GetBuffer()函数上。任意给定一个Python对象，该函数会尝试获取有关对象底层内存表示的相关信息。如果无法做到这点——大部分普通的Python对象都属于这种情况，则会产生一个异常并返回-1。传给PyBuffer_GetBuffer()的特殊标记进一步提供了所请求的内存缓冲区的类型。例如，PyBUF_ANY_CONTIGUOUS表示请求的是一段连续的内存。

针对数组、字节串以及其他类似的对象，结构体Py_buffer中会保存有关底层内存的信息。这包括一个指向内存块的指针、总内存大小、数组中每个元素的大小、格式以及其它细节。下面是这个结构体的定义：

```
typedef struct

bufferinfo {
    void

*buf;                  /* Pointer to buffer memory */
```

```
    PyObject *obj;             /* Python object that is the owne
    Py_ssize_t len;             /* Total size in bytes */
    Py_ssize_t itemsize;         /* Size in bytes of a single it
    int
readonly;             /* Read-only access flag */
    int
ndim;                 /* Number of dimensions */
    char
*format;             /* struct code of a single item */
    Py_ssize_t *shape;         /* Array containing dimensions */
    Py_ssize_t *strides;         /* Array containing strides */
    Py_ssize_t *suboffsets;     /* Array containing suboffsets */
```

```
} Py_buffer;
```

在本节中，我们只考虑接收一个内存连续的double型浮点数数组。要检查数组元素是否是double型的，可以检查format属性的格式化字符串是否为“d”。这个格式化字符串也是标准库中struct模块用来编码二进制值时所用的。一般来说，format可以是任意一种同struct模块相兼容的格式化字符串。如果数组中包含C结构体，那么这个格式化字符串也可能会包含多个类型代码。

一旦我们验证了底层缓冲区的信息，我们只需简单地将其传给C函数（示例中为avg()函数），则它会被当做一个普通的C数组来对待。这么做的实践意义在于不用考虑数组是什么类型，也不必考虑它是由什么库创建出来的。这就是为什么我们的函数既可以同array模块也可以同numpy库创建出的数组一起工作的原因。

在返回最终结果前，底层的缓冲区必须通过PyBuffer_Release()来释放。我们需要通过这个步骤来恰当地管理对象的引用计数。

再次申明，本节只展示了一小段接收数组的代码。如果要同数组打交道，则可能会遇到多维数组、不同的数据类型以及更多需要学习的技术。确保去查看官方文档（http://docs.python.org/3/c-api/buffer.html ）以获得更多的细节。

如果需要编写许多涉及数组处理的C扩展函数，可能会发现以Cython来实现这些代码会更容易些。具体请参见15.11节。

15.4　在C扩展模块中管理不透明指针

15.4.1　问题

我们有一个扩展模块需要处理指向C结构体的指针，但是不想把结构体的任何内部细节暴露给Python。

15.4.2　解决方案

不透明数据结构很容易通过将它们包装进一个capsule对象中来处理。考虑下面的代码片段：

```
typedef struct


Point {
    double


x,y;
} Point;

extern double


distance(Point *p1, Point *p2);
```

这里是一个扩展代码的示例，其中使用了capsule对象来对Point结构体和distance()函数进行包装：

```
/* Destructor function for points */

static void

del_Point(PyObject *obj) {
  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */

static

Point *PyPoint_AsPoint(PyObject *obj) {
  return

(Point *) PyCapsule_GetPointer(obj, "Point");
}

static

PyObject *PyPoint_FromPoint(Point *p, int

must_free) {
  return
```

```
PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}

/* Create a new Point object */



static


PyObject *py_Point(PyObject *self, PyObject *args) {
  Point *p;
  double


x,y;
  if


(!PyArg_ParseTuple(args,"dd",&x,&y)) {
    return


NULL;
  }
  p = (Point *) malloc(sizeof


(Point));
  p->x = x;
  p->y = y;
  return


PyPoint_FromPoint(p, 1);
}

static


PyObject *py_distance(PyObject *self, PyObject *args) {
  Point *p1, *p2;
  PyObject *py_p1, *py_p2;
  double
```

```
result;
  if


(!PyArg_ParseTuple(args,"OO",&py_p1, &py_p2)) {
    return


NULL;
  }
  if


(!(p1 = PyPoint_AsPoint(py_p1))) {
    return


NULL;
  }
  if


(!(p2 = PyPoint_AsPoint(py_p2))) {
    return


NULL;
  }
  result = distance(p1,p2);
  return


Py_BuildValue("d", result);
}
```

下面让我们从Python中来使用这些函数：

```
>>> import sample



>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1004ea330>
>>> p2
<capsule object "Point" at 0x1005d1db0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

15.4.3 讨论

capsule对象和C语言中的void指针很相似。capsule对象内部持有一个泛型指针以及一个标识名称。可以通过PyCapsule_New()函数来轻松创建出capsule对象。此外，可以在capsule对象上关联一个可选的析构函数，当capsule对象被垃圾收集机制回收时可用来释放底层的内存空间。

要提取出包含在capsule对象中的指针，可以通过函数PyCapsule_GetPointer()来完成，只要指定名称即可。如果提供的名称与capsule对象不匹配或者出现了其他的错误，那么会产生一个异常并返回NULL。

本节中我们编写了一对功能函数PyPoint_FromPoint()和PyPoint_AsPoint()，用来处理从capsule对象中创建和回退Point实例。在所有的扩展函数中，我们都会使用这一对函数而不是直接同capsule对象打交道。这个设计决策使得将来对包装Point对象的修改会变得更容易些。例如，如果稍后决定使用其他的机制而不是capsule对象的话，只需要修改这两个函数即可。

在使用capsule对象时，一个比较棘手的地方在于需要考虑垃圾收集和内存管理。函数PyPoint_FromPoint()接受一个must_free参数，表示当capsule对象被销毁时，底层的Point结构体所占用的内存是否也要被回收。当遇到这样的C代码时，对象的归属（ownership）问题会很难处理（例如，也许Point结构体嵌入到了另一个更大的数据结构中，而那个结构体是单独管理的）。与其把宝都押在垃圾收集上，这个额外的参数使得控制权重新回到程序员手上。应该要注意的是，已经在capsule对象上关联的析构函数也可以通过使用PyCapsule_SetDestructor()函数来修改。

当面对某些涉及结构体的C代码时，capsules是一种明智的解决方案。比如说，有时候我们并不在乎把结构体的细节暴露出来，或者会将其转换为一个全功能的扩展类型。有了capsule，我们可以为结

构体加上一个轻量级的包装层，这样可以轻松将其传递给其他的扩展函数。

15.5　在扩展模块中定义并导出C API

15.5.1　问题

我们有一个C扩展模块在内部定义了各种有用的函数，现在想将它们导出作为公有的C API在别处使用。我们想把这些函数用在其他的扩展模块中，但是不知道该如何将它们链接在一起，而用C编译器/链接器来做似乎又显得过于复杂（或者根本不可能做到）。

15.5.2　解决方案

本节把重点放在处理Point对象的代码上，代码在15.4节中已给出。如果回顾一下，这里的C代码中包含了一些实用的函数，比如：

```
/* Destructor function for points */

static void

del_Point(PyObject *obj) {
  free(PyCapsule_GetPointer(obj,"Point"));
```

```
}

/* Utility functions */



static


Point *PyPoint_AsPoint(PyObject *obj) {
  return


(Point *) PyCapsule_GetPointer(obj, "Point");
}

static


PyObject *PyPoint_FromPoint(Point *p, int


must_free) {
  return


PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}
```

现在的问题就是如何将函数PyPoint_AsPoint()和PyPoint_FromPoint()作为API导出，让其他的扩展模块可以使用和链接（例如，如果有其他的扩展模块也想使用包装过的Point对象）。

要解决这个问题，首先为示例扩展模块引入一个全新的头文件*pysample.h* 。将下列代码输入到这

个头文件中：

```
/* pysample.h */



#include "Python.h"
#include "sample.h"
#ifdef __cplusplus
extern


"C" {
#endif

/* Public API Table */



typedef struct


{
  Point *(*aspoint)(PyObject *);
  PyObject *(*frompoint)(Point *, int


);
} _PointAPIMethods;

#ifndef PYSAMPLE_MODULE
/* Method table in external module */



static


_PointAPIMethods *_point_api = 0;

/* Import the API table from sample */
```

```
static int


import_sample(void


) {
  _point_api = (_PointAPIMethods *) PyCapsule_Import("sample._poi
  return


(_point_api != NULL) ? 1 : 0;
}

/* Macros to implement the programming interface */



#define PyPoint_AsPoint(obj) (_point_api->aspoint)(obj)
#define PyPoint_FromPoint(obj) (_point_api->frompoint)(obj)
#endif

#ifdef __cplusplus
}
#endif
```

这里最重要的特性就是函数指针表 _PointAPIMethods。它会在导出模块中进行初始化，这样在导入模块中就可以找到它。

修改原来的扩展模块，增加这个函数指针表并像下面这样进行导出：

```
/* pysample.c */



#include "Python.h"
#define PYSAMPLE_MODULE
#include "pysample.h"

...
/* Destructor function for points */



static void


del_Point(PyObject *obj) {
  printf("Deleting point\n


");
  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */



static


Point *PyPoint_AsPoint(PyObject *obj) {
  return


(Point *) PyCapsule_GetPointer(obj, "Point");
}

static


PyObject *PyPoint_FromPoint(Point *p, int
```

```
free) {
  return


PyCapsule_New(p, "Point", free ? del_Point : NULL);
}

static _PointAPIMethods _point_api = {
  PyPoint_AsPoint,
  PyPoint_FromPoint
};
...

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
  PyObject *m;
  PyObject *py_point_api;

  m = PyModule_Create(&samplemodule);
  if (m == NULL)
    return NULL;

  /* Add the Point C API functions */
  py_point_api = PyCapsule_New((void *) &_point_api, "sample._poi
  if (py_point_api) {
    PyModule_AddObject(m, "_point_api", py_point_api);
   }
  return m;
}
```

最后，下面这个示例是一个新的扩展模块，它会加载并使用这些API函数：

```
/* ptexample.c */

/* Include the header associated with the other module */
```

```
#include "pysample.h"

/* An extension function that uses the exported API */



static


PyObject *print_point(PyObject *self, PyObject *args) {
  PyObject *obj;
  Point *p;
  if


(!PyArg_ParseTuple(args,"O", &obj)) {
    return


NULL;
  }

  /* Note: This is defined in a different module */



  p = PyPoint_AsPoint(obj);
  if


(!p) {
    return


NULL;
  }
  printf("%f %f\n


", p->x, p->y);
  return


Py_BuildValue("");
```

```
}

static


PyMethodDef PtExampleMethods[] = {
  {"print_point", print_point, METH_VARARGS, "output a point"},
  { NULL, NULL, 0, NULL}
};

static struct


PyModuleDef ptexamplemodule = {
  PyModuleDef_HEAD_INIT,
  "ptexample",                     /* name of module */



  "A module that imports an API",     /* Doc string (may be NULL



  -1,                             /* Size of per-interpreter sta



  PtExampleMethods                 /* Method table */



};

/* Module initialization function */



PyMODINIT_FUNC
PyInit_ptexample(void


) {
  PyObject *m;

  m = PyModule_Create(&ptexamplemodule);
```

```
  if

(m == NULL)
    return

NULL;

  /* Import sample, loading its API functions */

  if

(!import_sample()) {
    return

NULL;
  }

  return

m;
}
```

当编译这个新的模块时，我们甚至不必操心去链接任何库或者其他模块中的代码。只用创建一个简单的*setup.py* 文件即可：

```
# setup.py
```

```
from distutils.core import


setup, Extension

setup(name='ptexample',
      ext_modules=[
        Extension('ptexample',
                  ['ptexample.c'],
                  include_dirs = [], # May need pysample.h direct



                  )
        ]
)
```

如果一切顺利，就会发现我们的新扩展模块可以完美地同定义在其他模块中的C API一起工作了：

```
>>> import sample



>>> p1 = sample.Point(2,3)
>>> p1
<capsule object "Point *" at 0x1004ea330>
>>> import ptexample



>>> ptexample.print_point(p1)
2.000000 3.000000
>>>
```

15.5.3 讨论

本节所讨论的技术依赖于一个事实，即，capsule对象可以持有一个指针，该指针可指向任何所希望的对象。在这种情况下，定义capsule对象的模块会去填充函数指针结构体，创建一个capsule对象并让它指向这个函数指针表，最后将capsule对象保存在模块级属性中（即，sample._point_api）。

当导入模块后，其他的模块就可以通过编程的方式来获取这个属性并提取出底层的指针。实际上，Python提供了实用函数PyCapsule_Import()，它可以为我们完成所有的步骤。我们只用给它提供一个属性名（例如sample._point_api），它就会找到capsule对象并提取出指针。

这里用到了一些C编程技巧使得导出的函数在其他模块中看起来也并无不同之处。在文件*pysample.h* 中，指针_point_api用来指向在导出模块中初始化的函数指针表。用import_sample()函数来执行导入capsule对象以及初始化指针_point_api的任务。在使用模块中的其他函数之前，必须先调用import_sample()。通常这会在模块初始化的时候完成调用。最后，还定义了一组C预处理器宏以透明的方式通过函数指针来引用API函数。用户只需要使用原来的函数名即可，并不需要知道底层是通过

这些宏经过额外的一层间接关系来引用函数的。

最后，为什么要用这项技术来将模块链接在一起还有另一个重要的原因——这样做更加简单而且保证了模块间层次清晰、耦合度低。如果不想使用本节展示的技术，也可以利用共享库和动态加载器的功能来做交叉链接。比如，把所有公用的API函数放在共享库中，并确保所有的扩展模块都来链接这个共享库。是的，这么做可行，但是在大型系统中这么操作会非常繁琐。本质上，本节已经揭示了所有的魔法，允许模块通过Python的普通导入机制以及极少数的capsule调用实现对其他模块的链接。至于模块的编译问题，只需要担心头文件而不是共享库的实现细节。

更多有关为扩展模块提供 C API 的信息可以在 Python 文档（http://docs.python.org/3/extending/extending.html）中找到。

15.6　从C中调用Python

15.6.1　问题

我们想以安全的方式从C中执行一个Python的可调用对象，并将结果返回到C中。比方说，也许你正在编写C代码，希望把一个Python函数当做回调来使用。

15.6.2　解决方案

在C中调用Python基本上是简单明了的事，但是有几个地方需要用到一些技巧。下面的C代码作为一个示例展示了如何安全的从C中调用Python：

```
#include <Python.h>

/* Execute func(x,y) in the Python interpreter. The

   arguments and return result of the function must

   be Python floats */


double
```

```
call_func(PyObject *func, double

x, double

y) {
  PyObject *args;
  PyObject *kwargs;
  PyObject *result = 0;
  double

retval;

  /* Make sure we own the GIL */

  PyGILState_STATE state = PyGILState_Ensure();

  /* Verify that func is a proper callable */

  if

(!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n

");
    goto

fail;
  }
  /* Build arguments */

  args = Py_BuildValue("(dd)", x, y);
```

```
  kwargs = NULL;

  /* Call the function */



  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);

  /* Check for Python exceptions (if any) */



  if


(PyErr_Occurred()) {
    PyErr_Print();
    goto


fail;
  }

  /* Verify the result is a float object */



  if


(!PyFloat_Check(result)) {
    fprintf(stderr,"call_func: callable didn't return a float\n


");
    goto


fail;
  }

  /* Create the return value */
```

```
  retval = PyFloat_AsDouble(result);
  Py_DECREF(result);

  /* Restore previous GIL state and return */



  PyGILState_Release(state);
  return


retval;

fail:
  Py_XDECREF(result);
  PyGILState_Release(state);
  abort(); // Change to something more appropriate



}
```

要使用这个函数，需要将一个已存在的Python可调用对象的引用传递进来。有许多种方法可以实现，比如把一个可调用对象传递到一个扩展模块中，或者直接编写C代码从已有的模块中提取出相应的符号。

下面这个简单的例子展示了从一个嵌入的Python解释器中调用函数：

```
#include <Python.h>
```

```
/* Definition of call_func() same as above */



...
/* Load a symbol from a module */



PyObject *import_name(const char


*modname, const char


*symbol) {
  PyObject *u_name, *module;
  u_name = PyUnicode_FromString(modname);
  module = PyImport_Import(u_name);
  Py_DECREF(u_name);
  return


PyObject_GetAttrString(module, symbol);
}

/* Simple embedding example */



int


main() {
  PyObject *pow_func;
  double


x;

  Py_Initialize();
  /* Get a reference to the math.pow function */
```

```
  pow_func = import_name("math","pow");

  /* Call it using our call_func() code */



  for


(x = 0.0; x < 10.0; x += 0.1) {
    printf("%0.2f %0.2f\n


", x, call_func(pow_func,x,2.0));
  }
  /* Done */



  Py_DECREF(pow_func);
  Py_Finalize();
  return


0;
}
```

要构建这个最新的示例，需要编译上述C代码并同Python解释器链接。这里有一个Makefile告诉我们如何去做（可能需要在自己的机器上做些调整）

```
all::
        cc -g embed.c -I/usr/local/include/python3.3m \
```

```
        -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m
```

编译代码并运行得到的可执行文件，应该会产生类似这样的输出：

```
0.00 0.00
0.10 0.01
0.20 0.04
0.30 0.09
0.40 0.16
...
```

下面的示例稍有不同，一个扩展函数接收一个Python可调用对象以及一些参数，并将它们传递给call_func()用于测试：

```
/* Extension function for testing the C-Python callback */
PyObject *py_call_func(PyObject *self, PyObject *args) {
  PyObject *func;
  double x, y, result;
  if (!PyArg_ParseTuple(args,"Odd", &func,&x,&y)) {
    return NULL;
  }
  result = call_func(func, x, y);
  return Py_BuildValue("d", result);
}
```

使用这个扩展函数，可以像下面这样测试其功能：

```
>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>
```

15.6.3 讨论

如果要从C中调用Python，需要记住的最重要的事情就是此时C会获得程序的控制权。也就是说，创建参数、调用Python函数、检查是否有异常、检查类型、获取返回值等责任都落在了C的身上。

首先，很重要的一点是我们得有一个Python对象，用来代表打算去调用的那个可调用对象。这可以是函数、类、方法、内建方法或者任何实现了__call__()操作的对象。要验证对象是否是可调用的，可以使用下列代码片段中给出的PyCallable_Check()函数：

```
double call_func(PyObject *func, double x, double y) {
  ...
  /* Verify that func is a proper callable */
  if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
  }
  ...
```

顺便说一句，我们需要仔细学习如何在C代码中处理错误。一般来说，我们没法直接抛出一个Python异常。相反，错误需要按照C语言的方式来处理。在给出的解决方案中，我们使用goto来将控制流转移到一个错误处理块中，并在那里调用abort()函数。这会导致整个程序退出，但是在现实环境中可能需要做些更加优雅的处理（例如返回一个状态码）。请记住，此时是C代码在接管控制流，因此抛出异常是无法同C兼容的。错误处理必须由构建到程序中的组件来完成。

调用一个函数相对来说就简单直接多了——只需要调用PyObject_Call()，提供给它可调用对象、参数元组以及一个可选的关键字参数字典即可。要构建参数元组或者字典，可以使用Py_BuildValue()，示例如下：

```
double call_func(PyObject *func, double x, double y) {
  PyObject *args;
  PyObject *kwargs;

  ...
  /* Build arguments */
  args = Py_BuildValue("(dd)", x, y);
  kwargs = NULL;

  /* Call the function */
  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
```

```
  Py_XDECREF(kwargs);
  ...
```

如上述代码所示，如果没有关键字参数，那么可以传NULL。在完成函数调用后，需要确保使用Py_DECREF()或者Py_XDECREF()来清理参数。后者可安全地接受NULL指针（会忽略掉），这也是为什么我们用它来清理可选的关键字参数。

在调用了Python函数后，必须检查是否有异常出现。PyErr_Occurred()函数可以用来完成这个任务。比较棘手的地方在于知道如何去响应异常。因为我们工作在C语言的环境中，缺少Python所拥有的异常机制。因此，需要设定错误状态码，对错误做日志记录，或者去做一些明智的处理。在我们的解决方案中，由于没有更加简单的替代方案，因此直接调用了abort()（此外，C语言的拥护者也会更欣赏这种直接让程序崩溃的方案）。

```
  ...
  /* Check for Python exceptions (if any) */
  if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
  }
  ...
fail:
  PyGILState_Release(state);
  abort();
```

从调用的Python函数的返回值中提取出信息，一般来说需要涉及某种类型检查和提取值的过程。要做到这点，可能必须用到 Python concrete 对象层（https://docs. python.org/3/c-api/concrete.html ）中的函数。在解决方案中，我们使用了PyFloat_Check()和Py_Float_AsDouble()函数来检查并提取出Python浮点数。

关于从C中调用Python，最后一个棘手的部分在于管理Python的全局解释器锁（GIL）。每当从C中访问Python时，需要保证对GIL做合适的获取和释放动作。否则，就会有Python解释器破坏了数据或者崩溃的风险。调用PyGILState_Ensure()和PyGILState_Release()可确保正确地完成这些步骤：

```
double call_func(PyObject *func, double x, double y) {
  ...
  double retval;

  /* Make sure we own the GIL */
  PyGILState_STATE state = PyGILState_Ensure();
  ...
  /* Code that uses Python C API functions */
  ...
  /* Restore previous GIL state and return */
  PyGILState_Release(state);
  return retval;

fail:
  PyGILState_Release(state);
  abort();
}
```

调用PyGILState_Ensure()成功后，将总是保证调用线程对Python解释器享有独占访问权。甚至当调用的C代码正在运行着另一个对Python解释器来说未知的线程时也是如此。此时，C代码可自由使用任何想调用的Python C-API函数了。当成功返回时，使用PyGILState_Release()来将解释器恢复到它原来的状态。

要重点提到的是，每个PyGILState_Ensure()调用必须跟着一个匹配的PyGILState_Release()调用——甚至在出现错误的情况下也必须如此。在解决方案中，我们使用的goto语句可能看起来是种糟糕的设计，但是实际上我们利用它来将控制流跳转到一个公共的退出语句块中，在那里执行这个必要的步骤。可以把fail:标签后的代码想象成Python中的finally:语句块，它们的作用和目的是一样的。

如果我们编写的C代码使用了所有这些约定，包括管理GIL、检查异常以及对错误的彻底检查，将会发现我们能够以可靠的方式从C中调用Python解释器——即使在使用了高级编程技术（比如多线程）的复杂程序中也是如此。

15.7　在C扩展模块中释放GIL

15.7.1　问题

我们希望自己的C扩展代码能够同其他的线程一起在Python解释器中并发运行。要做到这点，需要释放并重新获取全局解释器锁（GIL）。

15.7.2　解决方案

在C扩展代码中，可以通过插入下列宏来释放并重新获取GIL：

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code.  Must not use Python API functions
   ...
   Py_END_ALLOW_THREADS
   ...
   return result;
}
```

15.7.3　讨论

GIL只能在一种情况下被安全的释放，即，如果可以保证在C代码中不执行任何Python C API函数。典型的例子就是在计算密集型代码中对C数组执行计算时（例如在numpy这样的扩展模块中）或者在执行阻塞式I/O操作的代码中（例如在文件描述符上执行读取或写入操作时）。

因为 GIL 的释放，其他Python线程就允许在解释器中执行了。宏Py_END_ALLOW_THREADS会阻塞执行，直到调用线程重新获取到GIL为止。

15.8 混合使用C和Python环境中的线程

15.8.1 问题

我们的程序中混合了C、Python和线程，但是其中有些线程是在C环境中创建的，不受Python解释器的控制。此外，还有一些特定的线程使用了Python C API中的函数。

15.8.2 解决方案

如果打算将C、Python和线程混合在一起使用，需要确保以恰当的方式初始化和管理Python的全局解释器锁（GIL）。要做到这点，可以在C代码中包含如下的代码，并确保在创建任何线程之前先调用它：

```
#include <Python.h>

  ...
  if


(!PyEval_ThreadsInitialized()) {
    PyEval_InitThreads();
  }
  ...
```

对于任何涉及Python对象或者Python C API的C代码，首先需要保证以合适的方式获取和释放GIL。这可以通过PyGILState_Ensure()和PyGILState_Release()来做到，示例如下：

```
...
/* Make sure we own the GIL */



PyGILState_STATE state = PyGILState_Ensure();

/* Use functions in the interpreter */




...
/* Restore previous GIL state and return */



PyGILState_Release(state);
...
```

每一个PyGILState_Ensure()调用都必须有一个配对的PyGILState_Release()。

15.8.3 讨论

在涉及C和Python的高级应用中，让许多事情同时运行的情况并非不常见——很可能会涉及C代码、Python代码、C线程以及Python线程。只要保证Python解释器经过恰当的初始化且涉及解释器相关的C代码能够管理好GIL，那么就可以正常工作。

请注意PyGILState_Ensure()并不会立刻抢占或中断解释器。如果其他代码正在执行中，这个函数将阻塞直到其他代码决定释放GIL为止。在内部，Python解释器会周期性地切换线程，因此即使另一个线程正在执行，调用方最终依然会得到运行（尽管可能先要等待一会儿）。

15.9 用Swig来包装C代码

15.9.1 问题

我们想将已有的C代码作为C扩展模块来访问。我们想通过Swig（http://www.swig.org ）来实现这一目标。

15.9.2 解决方案

Swig可以解析C头文件并自动创建出扩展代码来。要使用这个工具，首先需要有一个C头文件。例如，下面这个头文件就可用于我们的示例代码：

```
/* sample.h */



#include <math.h>
extern int


gcd(int


, int


);
extern int
```

```
in_mandel(double

x0, double

y0, int

n);
extern int

divide(int

a, int

b, int

*remainder);
extern double

avg(double

*a, int

n);

typedef struct

Point {
    double

x,y;
} Point;
```

```
extern double

distance(Point *p1, Point *p2);
```

一旦有了头文件，下一步就是编写一个Swig“接口”文件。根据约定，这些接口文件都以*.i*作为后缀，看起来类似于这样：

```
// sample.i - Swig interface


%module sample
%{
#include "sample.h"
%}

/* Customizations */


%extend Point {
    /* Constructor for Point objects */


    Point(double

x, double

y) {
        Point *p = (Point *) malloc(sizeof
```

```
(Point));
        p->x = x;
        p->y = y;
        return


p;
    };
};

/* Map int *remainder as an output argument */



%include typemaps.i
%apply int


*OUTPUT { int


* remainder };

/* Map the argument pattern (double *a, int n) to arrays */



%typemap(in) (double


*a, int


n)(Py_buffer view) {
  view.obj = NULL;
  if


(PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS | PyBUF_
    SWIG_fail;
  }
  if
```

```
(strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of double
    SWIG_fail;
  }
  $1 = (double


*) view.buf;
  $2 = view.len / sizeof


(double


);
}

%typemap(freearg) (double


*a, int


n) {
  if


(view$argnum.obj) {
    PyBuffer_Release(&view$argnum);
  }
}

/* C declarations to be included in the extension module */




extern int


gcd(int


, int
```

```
);
extern int

in_mandel(double

x0, double

y0, int

n);
extern int

divide(int

a, int

b, int

*remainder);
extern double

avg(double

*a, int

n);

typedef struct

Point {
    double
```

```
x,y;
} Point;

extern double

distance(Point *p1, Point *p2);
```

一旦编写好了这个接口文件，Swig就可以作为命令行工具在终端中调用了：

```
bash % swig -python -py3 sample.i
bash %
```

Swig会产生两个文件：*sample_wrap.c* 和 *sample.py* 。后者是用户用来导入的。*sample_wrap.c* 文件是C程序源代码，需要将其编译到一个支撑模块_sample中。这和普通的扩展模块所采用的技术一样。例如，要像这样创建一个*setup.py* 文件：

```
# setup.py

from distutils.core import

setup, Extension
```

```
setup(name='sample',
      py_modules=['sample.py'],
      ext_modules=[
        Extension('_sample',
                  ['sample_wrap.c'],
                  include_dirs = [],
                  define_macros = [],
                  undef_macros = [],
                  library_dirs = [],
                  libraries = ['sample']
                  )
        ]
)
```

要编译和测试，只要针对*setup.py* 文件运行python3即可：

```
bash % python3 setup.py build_ext --inplace
running build_ext
building '_sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-p
-I/usr/local/include/python3.3m -c sample_wrap.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o
sample_wrap.c: In function


‘SWIG_InitializeModule’:
sample_wrap.c:3589: warning: statement with no effect
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_
 build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -l
bash %
```

如果一切顺利，会发现现在能够直接使用得到

的C扩展模块了，示例如下：

```
>>> import sample



>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
>>> p1.y
3.0
>>> import array



>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>>
```

15.9.3 讨论

Swig是用来构建扩展模块的最古老的工具之一，时间要追溯到Python 1.4时期。但是，目前的版本已经开始对Python 3提供支持了。Swig的主要用途是使用Python作为高层控制语言来访问已有的大型C代码库。例如，用户的C代码中可能包含了

上千个函数以及各式各样的数据结构，而用户希望通过Python来访问它们。大部分生成包装函数的过程都可以用Swig来自动化进行。

所有的Swig接口都有着如下的简短开头：

```
%module sample
%{
#include "sample.h"
%}
```

这仅仅只是声明了扩展模块的名称，而且指定了必须要包含在内才能让所有组件通过编译的C头文件（包在%{和%}之间的代码会直接粘贴到输出的代码文件中，因此为了能编译通过，要将所有需要包含的头文件和其他的定义都放置在这里）。

Swig接口文件的底部是一些想包含在扩展模块中的C声明式。这些通常可以直接从头文件中拷贝过来。在我们的示例中，我们是直接从头文件中粘贴过来的：

```
%module sample
%{
#include "sample.h"
%}
...
extern int
```

```
gcd(int


, int


);
extern int


in_mandel(double


x0, double


y0, int


n);
extern int


divide(int


a, int


b, int


*remainder);
extern double


avg(double


*a, int


n);

typedef struct
```

```
Point {
    double

x,y;
} Point;

extern double

distance(Point *p1, Point *p2);
```

要重点强调的是这些声明式就是在告诉Swig你希望包含在Python模块中的内容。对声明列表做适当的修改和编辑是很常见的。比如，如果不想包含某些特定的声明式，可以从声明列表中将它们移除。

使用Swig最复杂的部分在于它可以对C代码做各种各样的定制化处理。这是个庞大的主题不可能在这里涵盖所有细节，但是本节中也展示了几个这样的定制化处理。

第一个定制化处理涉及%extend指令，它允许将方法关联到已有的结构体和类定义中。在示例中，我们使用这个技术给Point结构体添加了一个构造函数。这个定制化处理使得像这样使用结构体成为可能：

```
>>> p1 = sample.Point(2,3)
>>>
```

如果忽略这一步，那么Point对象就需要以更加复杂的方式来创建了：

```
>>> # Usage if %extend Point is omitted



>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3
```

第二个定制化处理涉及包含typemaps.i库，以及对%apply指令的使用。%apply指令告诉Swig参数签名int *remainder*应该被看做是一个输出值。这实际上是一个模式匹配规则。在后面所有的声明式中，只要遇到了*int* remainder，那么就把它当做输出处理。这个定制化处理使得divide()函数可以返回两个值：

```
>>> sample.divide(42,8)
[5, 2]
>>>
```

最后一个定制化处理涉及对%typemap指令的使用，这也许是本节所展示的最高级的功能了。typemap就是一种规则，可作用于输入中特定的参数模式上。在本节中，我们已经编写了一个typemap来匹配形式为(double *a, int n)这样的参数模式。在typemap内部是一段C代码，用来告诉Swig如何去把一个Python对象转换为相关的C参数。本节给出的代码中使用了Python中的buffer协议，用来对任何看起来像是double数组的输入参数做匹配（即，NumPy数组、由array模块创建的数组等）。对数组的操作可参见15.3节。

在typemap代码中，类似像$1和$2这样的替换符用来代表变量，这些变量保存着在typemap模式中经过转换的C参数的值（例如，$1会映射为double *a*，而*$2*会映射为*int n*）。*$input*表示一个*PyObject* 参数，它作为输入参数。$argument则表示参数的个数。

编写和理解typemap常常成为程序员使用Swig的最大障碍。不仅因为这种代码相当隐晦，而且需要我们同时对Python C API以及Swig与它们交互的方式的复杂细节有着很好的理解。Swig的文档中有更多的示例和详细的信息。

然而，如果有许多C代码需要以扩展模块的方

式暴露给Python，则Swig可以成为一件非常得力的工具。需要牢记的关键点就是Swig基本上就是一个用来处理C语言声明的编译器，但是还有着强大的模式匹配以及定制化组件，能让我们对特定的声明和类型的处理方式做出改变。更多信息可在 Swig 的网站（http://www.swig.org ）以及特定于Python的文档（http://www.swig.org/Doc2.0/Python.html ）中找到。

15.10　用Cython来包装C代码

15.10.1　问题

我们想用Cython来创建一个Python扩展模块，用来包装一个已有的C库。

15.10.2　解决方案

从某种程度上来看，用Cython创建一个扩展模块和手动编写扩展模块有些相似。它们都需要我们创建一组包装函数。但是与前几节不同的是，我们不必再用C来完成这些事情了——现在使用的代码看起来非常像Python。

提前说明，假设本章介绍部分给出的示例代码已经被编译为C库，名称为libsample。我们首先创建一个名为*csample.pxd* 的文件，它看起来是这样的：

```
# csample.pxd

#
```

```
# Declarations of "external" C functions and structures



cdef extern from "sample.h":
    int gcd(int, int)
    bint in_mandel(double, double, int)
    int divide(int, int, int *)
    double avg(double *, int) nogil

    ctypedef struct Point:
         double x
         double y

    double distance(Point *, Point *)
```

这个文件在Cython中的目的和作用就相当于一个C头文件。文件中最开始的声明cdef extern from "sample.h"声明了所需的C头文件。后面跟着的声明都是取自那个C头文件中。这个文件的名称是*csample.pxd*，不是*sample.pxd* ——这一点很重要。

接下来，创建一个名为*sample.pyx* 的文件。这个文件将定义包装函数，作为Python解释器到*csample.pxd* 文件中定义的底层C代码之间的桥梁：

```
# sample.pyx
```

```
# Import the low-level C declarations

cimport csample

# Import some functionality from Python and the C stdlib

from cpython.pycapsule cimport

*
from libc.stdlib cimport malloc

, free

# Wrappers

def

gcd(unsigned int x, unsigned int y):
    return

csample.gcd(x, y)

def

in_mandel(x, y, unsigned int n):
    return
```

```
csample.in_mandel(x, y, n)

def


divide(x, y):
    cdef int rem
    quot = csample.divide(x, y, &rem)
    return


quot, rem

def


avg(double[:] a):
    cdef:
        int sz
        double result

sz = a.size
with


nogil:
    result = csample.avg(<double *> &a[0], sz)
return


result

# Destructor for cleaning up Point objects



cdef del_Point(object obj):
    pt = <csample.Point *> PyCapsule_GetPointer(obj,"Point")
    free(<void *> pt)

# Create a Point object and return as a capsule
```

```
def


Point(double x,double y):
    cdef csample.Point *p
    p = <csample.Point *> malloc(sizeof(csample.Point))
    if


p == NULL:
        raise MemoryError


("No memory to make a Point")
    p.x = x
    p.y = y
    return


PyCapsule_New(<void *>p,"Point",<PyCapsule_Destructor>del_Point)

def


distance(p1, p2):
    pt1 = <csample.Point *> PyCapsule_GetPointer(p1,"Point")
    pt2 = <csample.Point *> PyCapsule_GetPointer(p2,"Point")
    return


csample.distance(pt1,pt2)
```

有关这个文件的各种细节将在讨论部分做进一步的说明。最后，要构建出扩展模块，需要创建一个*setup.py* 文件，看起来是这样的：

```
from distutils.core import
```

```
setup
from distutils.extension import


Extension
from Cython.Distutils import


build_ext

ext_modules = [
    Extension('sample',
              ['sample.pyx'],
              libraries=['sample'],
              library_dirs=['.'])]
setup(
  name = 'Sample extension module',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

要构建出最后的结果用于实验，在终端中键入如下命令：

```
bash % python3 setup.py build_ext --inplace
running build_ext
cythoning sample.pyx to sample.c
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-p
 -I/usr/local/include/python3.3m -c sample.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample.o
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_
 -L. -lsample -o sample.so
bash %
```

如果一切顺利，现在应该有一个名为sample.so的扩展模块了，可以像下列示例中那样使用：

```
>>> import sample



>>> sample.gcd(42,10)
2
>>> sample.in_mandel(1,1,400)
False
>>> sample.in_mandel(0,0,400)
True
>>> sample.divide(42,10)
(4, 2)
>>> import array



>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1005d1e70>
>>> p2
<capsule object "Point" at 0x1005d1ea0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

15.10.3 讨论

本节包含了一些在前面章节中讨论过的高级特性，这包括操作数组、包装不透明指针以及释放

GIL。这些部分在本节中会依次进行讨论，但我们先回顾一下前面的章节。

从高层来看，Cython是用来模仿C的。*.pxd* 文件仅仅只是包含了C中的声明（和C中的*.h* 文件类似），而*.pyx* 文件包含了实现（类似于*.c* 文件）。Cython中的cimport语句用来从*.pxd* 文件中导入声明。这和使用普通的Python import语句有所不同，在Python中这会加载一个正常的Python模块。

尽管*.pxd* 文件包含了声明，但它们并不是用来自动产生扩展代码的。因此，我们仍然必须编写简单的包装函数。例如，尽管*csample.pxd* 文件声明了函数int gcd(int, int)，我们仍然需要在*sample.pyx* 中编写一个包装函数。示例如下：

```
cimport csample

def


gcd(unsigned int x, unsigned int y):
    return


csample.gcd(x,y)
```

对于简单的函数，要做的事情并不会太多。

Cython会产生包装代码对参数和返回值做适当的转换。关联到参数上的C数据类型是可选的。但是，如果包含了它们，不用花任何代价就能获得额外的错误检查。例如，如果有人用负数做参数来调用这个函数，那么就会产生一个异常：

```
>>> sample.gcd(-10,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.pyx", line 7, in sample.gcd (sample.c:1284)
    def


gcd(unsigned int x,unsigned int y):
OverflowError: can't convert negative value to unsigned int
>>>
```

如果想给包装函数添加额外的检查机制，只要再增加包装的代码即可。示例如下：

```

def


gcd(unsigned int x, unsigned int y):
    if


x <= 0:
        raise ValueError


("x must be > 0")
    if
```

```
y <= 0:
        raise ValueError


("y must be > 0")
    return


csample.gcd(x,y)
```

在*csample.pxd* 文件中声明的in_mandel()有一个有趣但不太明显的地方。在那个文件中，函数被声明为返回一个bint而不是int。这使得函数会创建一个合适的布尔值而不是一个简单的整型值。因此，返回值0会被映射为False，而1会对应True。

在Cython包装函数内，除了可以使用所有普通的Python对象外，还可以选择声明C数据类型。divide()的包装函数就展示了这样一个用法，也演示了如何处理指针参数。

```
def


divide(x,y):
    cdef int rem
    quot = csample.divide(x,y,&rem)
    return
```

```
quot, rem
```

这里，变量rem被显式声明为C语言中的int变量。当传递给下面的divide()函数时，&rem就表示指向rem的指针，这和C语言中的表达方式一致。

avg()函数的代码则说明了Cython中一些更加高级的功能。首先，声明式def avg(double[:] a)将avg()声明为接受一个一维double值的内存视图（memoryview）。令人惊讶的地方在于这使得avg函数将接受任意一种兼容的数组对象，包括那些由numpy库所创建的数组也是如此。示例如下：

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> import numpy
>>> b = numpy.array([1., 2., 3.])
>>> import sample
>>> sample.avg(a)
2.0
>>> sample.avg(b)
2.0
>>>
```

在包装函数中，a.size和&a[0]分别表示数组元素的个数以及指向数组的指针。语法<double *> &a[0]可以让我们在必要的时候对指针类型进行转换。需要保证C中的avg()函数接受的是类型正确的指针。下一节中会介绍一些有关Cython中内存视图的高级用法。

除了同常规数组打交道外，avg()的示例也展示了如何同全局解释器锁（GIL）打交道。语句with nogil:声明了一个代码块，表示执行时不持有GIL。在语句块内部，使用任何普通的Python对象都是非法的——只有声明为cdef的对象和函数可以使用。除此之外，外部函数必须显式声明为可以不持有GIL执行。因此，在*csample.pxd* 文件中函数avg()被声明为double avg(double *, int) nogil。

对结构体Point的处理则带来了特殊的挑战。如前文所示，本节使用capsule对象将Point对象当做不透明指针来处理，这部分内容在15.4节中已有说明。但是，要做到这点，底层的Cython代码就要更加复杂一点。首先，下面这些导入语句是用来从C库和Python C API中引入函数定义：

```
from cpython.pycapsule cimport *
from libc.stdlib cimport malloc, free
```

函数del_Point()和Point()使用导入的函数来创建一个capsule对象用来包装Point *指针。声明式cdef del_Point()把del_Point()声明为一个只能从Cython而不是Python中访问的函数。因而，这个函数对外部是不可见的——相反，它作为回调函数来清理由capsule对象占用的内存空间。对PyCapsule_New()和PyCapsule_GetPointer()的调用是直接从Python C API中得来的，使用方法一样。

distance()函数会从由Point()中创建出的capsule对象里提取出指针。这里值得注意的是我们不必担心异常处理的问题。如果传递了一个不正确的对象，PyCapsule_GetPointer()会产生一个异常，但是Cython会去查找异常并将其传播到distance()外部。

在这个解决方案中，对Point结构体的处理缺点在于这完全是不透明的。我们无法查看或访问任何属性。下面还有一个替代方案，就是定义一个扩展类型，示例如下：

```
# sample.pyx

cimport csample
```

```
from libc.stdlib cimport malloc


, free



...

cdef class Point


:
    cdef csample.Point *_c_point
    def


__cinit__(self, double x, double y):
        self._c_point = <csample.Point *> malloc(sizeof(csample.P
        self._c_point.x = x
        self._c_point.y = y

    def


__dealloc__(self):
        free(self._c_point)

    property x:
        def


__get__(self):
            return


self._c_point.x
        def


__set__(self, value):
            self._c_point.x = value

    property y:
        def
```

```
__get__(self):
            return


self._c_point.y
        def


__set__(self, value):
            self._c_point.y = value

def


distance(Point p1, Point p2):
    return


csample.distance(p1._c_point, p2._c_point)
```

这里，cdef class Point将Point声明为一个扩展类型。类变量cdef csample.Point *_c_point用来保存指向底层C代码中Point结构体的指针。__cinit__()和__dealloc__()方法使用malloc()和free()调用来创建和销毁底层的C结构体。property x和property y声明让代码可以对底层的结构体属性进行get和set操作。distance()的包装函数也被适当地修改为接受Point扩展类型实例作为参数，但是会传递底层的指针给C函数。

做了这样的修改，会发现现在用来操作Point对

象的代码会更加自然些：

```
>>> import sample



>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<sample.Point object at 0x100447288>
>>> p2
<sample.Point object at 0x1004472a0>
>>> p1.x
2.0
>>> p1.y
3.0
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

本节介绍了许多Cython的核心功能，我们可以将它们应用到情况更加复杂的包装处理上。可以肯定的是，要想实现更多功能，一定要去看看官方文档（http://docs.cython.org ）。

接下来的几节同样会介绍一些额外的Cython功能。

15.11　用Cython来高效操作数组

15.11.1　问题

我们想编写一些用来处理数组的高性能函数，把它们作用在类似NumPy这样的库创建的数组上。我们听说像Cython这样的工具会让这个过程变得简单，但是不确定该如何去做。

15.11.2　解决方案

作为示例，下面的代码展示了一个用来对一维double数组元素进行修改的函数：

```
# sample.pyx (Cython)

cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    '''
    Clip the values in a to be between min and max. Result in out
    '''
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the sar
    for i in range(a.shape[0]):
        if a[i] < min:
            out[i] = min
        elif a[i] > max:
```

```
            out[i] = max
        else:
            out[i] = a[i]
```

要编译并构建这个扩展函数，需要一个下面这样的*setup.py* 文件（使用命令python3 setup.py build_ext --inplace来构建）：

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',
              ['sample.pyx'])
]

setup(
  name = 'Sample app',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

就会发现得到的函数可以对数组元素进行修改，而且可以用于许多不同类型的数组对象。例如：

```
>>> # array module example
>>> import sample
>>> import array
>>> a = array.array('d',[1,-3,4,7,2,0])
>>> a
```

```
array('d', [1.0, -3.0, 4.0, 7.0, 2.0, 0.0])
>>> sample.clip(a,1,4,a)
>>> a
array('d', [1.0, 1.0, 4.0, 4.0, 2.0, 1.0])

>>> # numpy example
>>> import numpy
>>> b = numpy.random.uniform(-10,10,size=1000000)
>>> b
array([-9.55546017, 7.45599334, 0.69248932, ..., 0.69583148,
       -3.86290931, 2.37266888])
>>> c = numpy.zeros_like(b)
>>> c
array([ 0., 0., 0., ..., 0., 0., 0.])
>>> sample.clip(b,-5,5,c)
>>> c
array([-5. , 5. , 0.69248932, ..., 0.69583148,
       -3.86290931, 2.37266888])
>>> min(c)
-5.0
>>> max(c)
5.0
>>>
```

同样，我们也会发现得到的代码运行起来很快。在下面的交互式会话中，我们用这个实现同numpy库中已有的clip()函数进行了一场针锋相对的比拼：

```
>>> timeit('numpy.clip(b,-5,5,c)','from __main__ import b,c,numpy
8.093049556000551
>>> timeit('sample.clip(b,-5,5,c)','from __main__ import b,c,samp
...           number=1000)
3.760528204000366
>>>
```

可以看到，我们的实现要快上许多——考虑到NumPy的版本其核心是用C语言编写的，得出这样的结果真是有趣。

15.11.3　讨论

本节利用了Cython中的类型化内存视图（typed memoryview），它极大地简化了操作于数组的代码。语句cpdef clip()将clip()同时声明为C和Python函数。在Cython中这么做是很有用的，因为这意味着该函数在其他Cython函数中调用起来要更有效率（例如，如果想从另一个不同的Cython函数中调用clip()）。

类型化参数double[:] a以及double[:] out将这些参数声明为一维double数组。作为输入，它们可以访问任何实现了内存视图接口的数组对象（有关内存视图接口，可参见PEP 3118）。这包括NumPy以及内建的array库中的数组。

如果编写的代码产生的结果同样是数组，我们应该遵循示例代码中所展示的惯例，即，让一个参数成为输出参数。这就把创建输出数组的责任留给了调用者，而实现者则不必了解太多有关数组是什么类型的具体细节（这里只假设数组已经准备就位，只需要做一些基本的检查，例如确保它们的大

小是兼容的）。在NumPy这样的库中，使用numpy.zeros()或者numpy.zeros_like()来创建输出数组相对来说是很容易的。或者，要创建未初始化的数组，可以使用numpy.empty()或者numpy.empty_like()。如果打算用结果来覆盖数组内容，这么做会稍微更快些。

在函数的实现中，只需要利用索引（例如a[i]、out[i]等）编写简单直接的代码来处理数组即可。Cython会采取措施确保产生高效的代码。

位于clip()定义之前的那两个装饰器是用来做性能优化的可选项。@cython.boundscheck (False)会消除所有的数组边界检查，如果已经知道索引不会越界，那么就可以使用它。@cython.wraparound(False)在包装整个数组时会消除针对下标为负数时的处理。包含了这些装饰器能让代码的运行速度显著提高（对于这个示例，我们测试的结果是会快大约2.5倍）。

每当同数组打交道时，对底层的算法做仔细的研究和试验同样也能获得巨大的速度提升。例如，考虑下面这个clip()函数的变种，这里使用了条件表达式：

```
@cython.boundscheck(False)
@cython.wraparound(False)
```

```
cpdef clip(double[:] a, double min, double max, double[:] out):
    if


min > max:
        raise ValueError


("min must be <= max")
    if


a.shape[0] != out.shape[0]:
        raise ValueError


("input and output arrays must be the same size")
    for


i in


range(a.shape[0]):
        out[i] = (a[i] if


a[i] < max else


max) if


a[i] > min else


min
```

测试的时候，这个版本的代码运行速度又要快50%多（在timeit()测试中，成绩是2.44秒对比之前

的3.76秒）。

此时，我们可能想知道为什么这份代码在执行效率上会力压手写的C语言版本。例如，也许我们编写了如下的C函数，然后利用前面几节中谈到的技术手工编写了一个C语言扩展版本：

```
void


clip(double


*a, int


n, double


min, double


max, double


*out) {
  double


x;
  for


(; n >= 0; n--, a++, out++) {
    x = *a;
    *out = x > max ? max : (x < min ? min : x);
  }
}
```

这里并没有给出扩展代码。但是经过测试后，我们发现手工打造的C扩展代码却比由Cython创建出的扩展函数要慢10%。底线就是，Cython生成的代码运行速度比想象的还要快很多。

解决方案中给出的代码还可以做几个扩展。针对特定类型的数组操作，释放GIL使得多个线程能够并行运行是很有意义的。为了做到这点，修改代码使其包含with nogil:语句：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if


min > max:
        raise ValueError


("min must be <= max")
    if


a.shape[0] != out.shape[0]:
        raise ValueError


("input and output arrays must be the same size")
    with


nogil:
        for
```

```
i in

range(a.shape[0]):
            out[i] = (a[i] if

a[i] < max else

max) if

a[i] > min else

min
```

如果想编写一个能操作二维数组的版本，下面是一种解决方案：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip2d(double[:,:] a, double min, double max, double[:,:] o
    if

min > max:
        raise ValueError

("min must be <= max")
    for

n in
```

```
range(a.ndim):
        if


a.shape[n] != out.shape[n]:
            raise TypeError


("a and out have different shapes")
    for


i in


range(a.shape[0]):
        for


j in


range(a.shape[1]):
            if


a[i,j] < min:
                out[i,j] = min
            elif


a[i,j] > max:
                out[i,j] = max
            else


:
                out[i,j] = a[i,j]
```

希望读者看到这里时不会感到迷惑。本节所有给出的代码都不会特定于任何一种数组库（例如NumPy）。这使得代码有着非常好的灵活性。但是同样值得注意的是，一旦牵涉到多维数组、步进、偏移以及其他的因素，那么对数组的处理就会变得更加复杂。这些主题超出了本节的范围，但是更多信息可以在PEP 3118（http://www.python.org/dev/peps/pep-3118 ）中找到。Cython文档中关于类型化内存视图（http://docs.cython.org/src/userguide/memoryviews.html ）的章节也同样值得阅读。

15.12 把函数指针转换为可调用对象

15.12.1 问题

我们已经获取了某个C函数的内存地址，但是想将其转换成一个Python的可调用对象，这样我们可以把它当做扩展函数使用。

15.12.2 解决方案

ctypes模块可用来创建包装任意内存地址的可调用对象。下面的示例展示了如何获取一个C函数的底层原始地址，以及如何将其转换成一个可调用对象：

```
>>> import ctypes



>>> lib = ctypes.cdll.LoadLibrary(None)
>>> # Get the address of sin() from the C math library



>>> addr = ctypes.cast(lib.sin, ctypes.c_void_p).value
>>> addr
140735505915760
```

```
>>> # Turn the address into a callable function



>>> functype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double
>>> func = functype(addr)
>>> func
<CFunctionType object at 0x1006816d0>

>>> # Call the resulting function



>>> func(2)
0.9092974268256817
>>> func(0)
0.0
>>>
```

15.12.3 讨论

要创建一个可调用对象，必须首先创建一个CFUNCTYPE实例。CFUNCTYPE()的第一个参数是返回类型，接下来的参数就是原始函数的参数类型。一旦定义了函数类型，就用它来包装一个整数内存地址以此创建出一个可调用对象。得到的结果对象就可以通过ctypes像任何普通函数一样进行访问了。

本节讨论的内容可能看起来相当隐晦也非常底层。然而，对于程序和库来说，利用像LLVM库中使用的即时编译（just in-time compilation）这类高

级代码生成技术也变得越来越普遍了。

例如，下面这样一个简单的示例使用llvmpy扩展（http://www.llvmpy.org ）来创建一个小的汇编函数，获取一个指向该函数的指针，然后将其转换为一个Python可调用对象：

```
>>> from llvm.core import


Module, Function, Type, Builder
>>> mod = Module.new('example')
>>> f = Function.new(mod,Type.function(Type.double(), \
                        [Type.double(), Type.double()], False),
>>> block = f.append_basic_block('entry')
>>> builder = Builder.new(block)
>>> x2 = builder.fmul(f.args[0],f.args[0])
>>> y2 = builder.fmul(f.args[1],f.args[1])
>>> r = builder.fadd(x2,y2)
>>> builder.ret(r)
<llvm.core.Instruction object at 0x10078e990>
>>> from llvm.ee import


ExecutionEngine
>>> engine = ExecutionEngine.new(mod)
>>> ptr = engine.get_pointer_to_function(f)
>>> ptr
4325863440
>>> foo = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double, cty

>>> # Call the resulting function



>>> foo(2,3)
13.0
>>> foo(4,5)
41.0
>>> foo(1,2)
```

```
5.0
>>>
```

不用多说，如果在这个层次上出任何差错的话将导致Python解释器“死于非命”。请记住，我们正在直接同机器的内存地址和原生机器码打交道，而不是Python函数。

15.13　把以NULL结尾的字符串传给C库

15.13.1　问题

我们正在编写的扩展模块需要把以NULL结尾的字符串传给C库。但是，我们并不能完全确定如何通过Python的Unicode字符串实现来做到这点。

15.13.2　解决方案

许多C库中包含的函数都把以NULL结尾的字符串类型声明为char *。考虑如下的C函数，我们将用它作为说明和测试的例子：

```
void

print_chars(char

*s) {
    while

(*s) {
        printf("%2x ", (unsigned char
```

```
) *s);
        s++;
    }
    printf("\n


");
}
```

这个函数只是简单地打印出每个字符的十六进制表示，这样可以方便地对传入的字符串进行调试。例如：

```
print_chars("Hello"); // Outputs: 48 65 6c 6c 6f
```

要从Python中调用这样一个C函数，有好几种选择。第一，可以使用转换代码“y”限制函数PyArg_ParseTuple()只操作字节，示例如下：

```

static


PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char


*s;

  if
```

```
(!PyArg_ParseTuple(args, "y", &s)) {
    return


NULL;
  }
  print_chars(s);
  Py_RETURN_NONE;
}
```

得到的结果函数可以像下面这样进行操作。仔细观察嵌入了NULL字节的字节流是如何处理的，以及传入Unicode字符串时会被拒绝执行：

```
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00


World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes without null bytes, not bytes
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
>>>
```

如果想传入Unicode字符串，可以使用格式化代码“s”，示例如下：

```
static


PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char


*s;

  if


(!PyArg_ParseTuple(args, "s", &s)) {
    return


NULL;
  }
  print_chars(s);
  Py_RETURN_NONE;
}
```

当使用上面的函数时，会自动将所有的字符串转换为以NULL结尾且以UTF-8编码的形式。示例如下：

```
>>> print_chars('Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars('Spicy Jalape\u00f1


o') # Note: UTF-8 encoding



53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_chars('Hello\x00
```

```
World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str without null characters, not str
>>> print_chars(b'Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>>
```

如果基于某些原因，我们需要直接同PyObject *打交道而不能使用PyArg_ParseTuple()，下面的代码示例告诉我们如何从字节流和字符串对象中检查并提取出一个合适的char *引用：

```
/* Some Python Object (obtained somehow) */



PyObject *obj;

/* Conversion from bytes */



{
   char



*s;
   s = PyBytes_AsString(o);
   if



(!s) {
```

```
        return


NULL; /* TypeError already raised */



    }
    print_chars(s);
}

/* Conversion to UTF-8 bytes from a string */



{
    PyObject *bytes;
    char


*s;
    if


(!PyUnicode_Check(obj)) {
        PyErr_SetString(PyExc_TypeError, "Expected string");
        return


NULL;
    }
    bytes = PyUnicode_AsUTF8String(obj);
    s = PyBytes_AsString(bytes);
    print_chars(s);
    Py_DECREF(bytes);
}
```

上面这两种转换都可保证接受以NULL结尾的数据，但是它们没有检查是否在字符串中间插入了

NULL字节的情况。因此，如果这对我们而言很重要的话，那就需要自己做检查了。

15.13.3　讨论

如果可能的话，应该避免编写出依赖以NULL结尾的字符串的代码，因为Python对字符串并没有这样的要求。如果可能的话，处理字符串时使用指针并配合使用表示其大小的参数几乎总是更好的选择。然而，有时候我们不得不去处理遗留的C代码，那样的话就别无选择了。

尽管上述技术很容易使用，但在PyArg_ParseTuple()中使用格式化代码“s”时会有一些内存方面的开销，而这是很容易被忽略的地方。当编写代码时，如果使用了上述转换规则，则会创建一个UTF-8编码的字符串，并且会永久将其关联到原始的字符串对象上。如果原始字符串中包含有非ASCII字符，那么就会使得字符串的大小增加，直到被垃圾收集机制处理为止。示例如下：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s) # Passing string
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s) # Notice increased size
103
>>>
```

如果对这种内存使用的增加需要纳入考虑，应该使用PyUnicode_AsUTF8String()函数来重新编写C扩展代码，示例如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *o, *bytes;
  char *s;

  if (!PyArg_ParseTuple(args, "U", &o)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(o);
  s = PyBytes_AsString(bytes);
  print_chars(s);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

按照上面这样修改，UTF-8编码的字符串会根据需要进行创建，但是会在使用完毕之后丢弃。下面是修改过的版本产生的行为，可以看到字符串的大小并没有增加：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
87
>>>
```

如果打算把以NULL结尾的字符串传递给通过ctypes包装的函数，请注意ctypes只允许传入字节，而且不会检查传入的字节中是否有插入NULL。示例如下：

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary("./libsample.so")
>>> print_chars = lib.print_chars
>>> print_chars.argtypes = (ctypes.c_char_p,)
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
48 65 6c 6c 6f
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
>>>
```

如果想传入字符串而不是字节流，则需要手动先执行一次UTF-8编码。示例如下：

```
>>> print_chars('Hello World'.encode('utf-8'))
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>>
```

对于其他的C扩展模块编写工具（例如Swig、Cython），是否要使用它们将字符串传递给C代码

则需要做仔细的研究。

15.14 把Unicode字符串传递给C库

15.14.1 问题

我们正在编写的扩展模块需要把Python字符串传递给一个C语言编写的库函数，而这个库函数可能并不知道该如何恰当地处理Unicode。

15.14.2 解决方案

这里需要考虑的问题比较多，但是主要问题在于已有的C库并不能理解Python原生的Unicode字符串表示。因此，我们的挑战就是将Python字符串转换为让C库可以更容易理解的形式。

为了更好地说明这个问题，下面给出了两个C函数。出于调试和实验的目的，这两个函数操作字符串数据并产生输出。其中一个函数使用的是以char *, int形式给出的字节流，而另一个函数使用的是以wchar_t *, int形式给出的宽字符。示例如下：

```
void print_chars(char *s, int len) {
  int n = 0;
  while (n < len) {
```

```
    printf("%2x ", (unsigned char) s[n]);
    n++;
  }
  printf("\n");
}

void print_wchars(wchar_t *s, int len) {
  int n = 0;
  while (n < len) {
    printf("%x ", s[n]);
    n++;
  }
  printf("\n");
}
```

对于操作字节流的函数print_chars()来说，需要将Python字符串转换成一个合适的字节编码，例如UTF-8。示例如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

对于可以处理机器上原生的wchar_t类型的库函数来说，可以像这样编写扩展代码：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args)
```

```
  wchar_t *s;
  Py_ssize_t len;

  if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
    return NULL;
  }
  print_wchars(s,len);
  Py_RETURN_NONE;
}
```

下面的交互式会话说明了这些函数是如何工作的：

```
>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>
```

请仔细观察基于字节流的函数print_chars()是如何接受UTF-8编码的数据的，而print_wchars()是如何接受Unicode码点值的（Unicode code point value）。

15.14.3　讨论

在正式讨论前，应该先研究一下我们打算访问的C函数库有哪些本质特性。对于许多C库来说，传递字节流比传递字符串要显得更有意义些。要做

到这点，可以使用下面的转换代码：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  /* accepts bytes, bytearray, or other byte-like object */
  if (!PyArg_ParseTuple(args, "y#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

如果仍然决定传递字符串，我们需要了解到Python 3使用的字符串表示虽然适应性很强，但并不能全部直接映射到使用标准的char *或wchar_t *类型的C库中去，有关这方面的细节可参考PEP 393（http://www.python.org/dev/peps/pep-0393 ）。因此，要把字符串数据传递给C，那么几乎总是要做某种形式的转换才行。在PyArg_ParseTuple()中使用格式化代码s#和u#可以安全地执行这样的转换。

可能存在的缺点就是这样的转换会导致原始字符串对象的大小永久性的增加。每当进行转换时，经过转换的数据会做一份拷贝关联到原始字符串对象上，这样稍后可以得到重用。可以通过下面的交互式会话来观察这个效果：

```
>>> import sys
```

```
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>
```

如果字符串数据总量很小，这种开销就无关紧要。但是如果需要在扩展模块中进行大量的文本处理，很可能就希望能够避免这种开销。下面对第一个扩展函数给出了另一种替代实现，这里就避免了这些内存开销：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char *s;
  Py_ssize_t len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(obj);
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

要避免wchar_t处理的内存开销则更为棘手。在内部，Python使用最有效的表示来存储字符串。例如，使用字节数组来存储仅包含ASCII的字符串，但是，如果字符串包含的字符范围是在U+0000~U+FFFF之间，则需要使用两个字节来表示。由于不存在数据的单一表示，因此无法将内部数组转换为wchar_t *，更别指望它会奏效。相反，必须创建一个wchar_t数组，然后将文本复制进来。PyArg_ParseTuple( )的“u#”格式代码会执行该操作，但是会以牺牲效率为代价（它将生成的副本附加到字符串对象）。

如果想避免这一长期的内存开销，唯一的选择是将Unicode数据复制到一个临时的数组，将其传递给C库函数，然后释放该数组。下面是一个可能的实现：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args)
PyObject *obj;
wchar_t *s;
Py_ssize_t len;

if (!PyArg_ParseTuple(args, "U", &obj)) {
return NULL;
}
if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {
return NULL;
}
print_wchars(s, len);
PyMem_Free(s);
Py_RETURN_NONE;
}
```

在这个实现中，PyUnicode_AsWideCharString()针对wchar_t字符创建了一个临时的缓冲区，并将数据拷贝到这里。这个缓冲区会在传递给C之后释放掉。在写作本小节时，关于这个行为似乎有一个bug，可以在Python的问题页面（bugs.python.org/issue16254）中找到相关的描述。

如果出于某些原因知道C库会以其他非UTF-8编码的形式来处理数据，我们可以强制Python执行适当的转换，这可以通过下面的扩展函数来完成：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s = 0;
  int len;
  if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len))
    return NULL;
  }
  print_chars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

最后但同样重要的是，如果想直接同Unicode字符串中的字符打交道，下面给出的示例说明了其中的底层访问机制：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args)
```

```
  PyObject *obj;
  int n, len;
  int kind;
  void *data;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if (PyUnicode_READY(obj) < 0) {
    return NULL;
  }
  len = PyUnicode_GET_LENGTH(obj);
  kind = PyUnicode_KIND(obj);
  data = PyUnicode_DATA(obj);

  for (n = 0; n < len; n++) {
    Py_UCS4 ch = PyUnicode_READ(kind, data, n);
    printf("%x ", ch);
  }
  printf("\n");
  Py_RETURN_NONE;
}
```

在上述代码中，宏PyUnicode_KIND()和PyUnicode_DATA()是同Unicode的宽度可变存储相关的，在PEP 393中可找到相关描述。变量kind对底层的存储信息（8位、16位或32位）进行编码，而变量data则指向缓冲区。事实上，只要把它们传递给PyUnicode_READ()宏即可，不需要再做任何处理。

最后再多说几句，当从Python中把Unicode字符串传递给C时，应该尽可能选择简单的方案。如果在UTF-8编码和宽字符中选择，那就选UTF-8。提

供对UTF-8的支持似乎更加普遍，麻烦也会少些，解释器对UTF-8的支持也更好。最后，请务必查看处理Unicode的相关文档（https://docs.python.org/3/c-api/Unicode.html ）。

15.15　把C字符串转换到Python中

15.15.1　问题

我们想把C字符串转换为Python字节流或者字符串对象。

15.15.2　解决方案

对于以char *, int形式表示的C字符串，我们必须决定是否要将它们以原始字节串或者Unicode字符串的形式来表示。字节对象可以通过Py_BuildValue()来构建，示例如下：

```
char *s;    /* Pointer to C string data */
int len;    /* Length of data */

/* Make a bytes object */
PyObject *obj = Py_BuildValue("y#", s, len);
```

如果想构建一个Unicode字符串，而且知道s指向的数据是以UTF-8来编码的，则可以使用如下的代码来完成：

```
   PyObject *obj = Py_BuildValue("s#", s, len);
```

如果s指向的数据是以其他已知的格式来编码的，可以通过PyUnicode_Decode()来创建字符串对象：

```
PyObject *obj = PyUnicode_Decode(s, len, "encoding", "errors");

/* Examples /*
obj = PyUnicode_Decode(s, len, "latin-1", "strict");
obj = PyUnicode_Decode(s, len, "ascii", "ignore");
```

如果刚好有一个以wchar_t *, len表示的宽字符串，这里就有一些选择。首先，可以像这样使用Py_BuildValue()：

```
wchar_t *w;   /* Wide character string */
int len;      /* Length */

PyObject *obj = Py_BuildValue("u#", w, len);
```

其次，可以直接使用PyUnicode_FromWideChar()：

```
PyObject *obj = PyUnicode_FromWideChar(w, len);
```

对于宽字符串来说，不会对字符数据做任何解

释——这里会假设把原始的Unicode码点直接转换到Python中。

15.15.3 讨论

把字符串从C转换到Python所遵循的准则同I/O一样。即，C中的数据必须依据某个编码规则显式将其解码为字符串。常见的编码包括ASCII、Latin-1以及UTF-8。如果不能百分之百地确定编码格式或者数据本身是二进制的，那么最好的选择就是将字符串编码为字节流。

当创建对象时，Python总是会拷贝我们提供的字符串数据。如果有必要的话，之后释放C字符串的任务就落在我们的身上。此外，为了获得更好的可靠性，在创建字符串时应该同时使用指针和表示字符串大小的参数，而不是依赖于以NULL结尾的数据。

15.16　同编码方式不确定的C字符串打交道

15.16.1　问题

我们需要在C和Python之间来回转换字符串，但是字符串在C中的编码方式是不确定的或者说是未知的。例如，假设C中的数据应该是按照UTF-8来编码的，但这个约定并没有得到强制执行。我们想编写代码能够以优雅的方式处理有问题的数据，在处理的过程中不会导致Python崩溃，也不会破坏字符串数据。

15.16.2　解决方案

下面的C函数以及数据可用来说明这个问题的本质：

```
/* Some dubious string data (malformed UTF-8) */

const char

*sdata = "Spicy Jalape\xc3\xb1
```

```
o\xae

";
int

slen = 16;

/* Output character data */

void

print_chars(char

*s, int

len) {
  int

n = 0;
  while

(n < len) {
    printf("%2x ", (unsigned char

) s[n]);
    n++;
  }
  printf("\n

");
}
```

在上述代码中，字符串sdata将UTF-8和格式不正确的数据混合在了一起。然而，如果用户在C中调用print_chars(sdata, slen)，却能够正常工作。

现在假设我们想将sdata的内容转换为Python字符串。进一步假设我们想稍后再把这个Python字符串通过扩展模块传回给print_chars()函数。下面给出的做法可以保证就算有编码问题存在，也能够完全保留原始数据不变。

```
/* Return the C string back to Python */

static

PyObject *py_retstr(PyObject *self, PyObject *args) {
  if

(!PyArg_ParseTuple(args, "")) {
    return

NULL;
  }
  return

PyUnicode_Decode(sdata, slen, "utf-8", "surrogateescape");
}

/* Wrapper for the print_chars() function */
```

```
static


PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char


*s = 0;
  Py_ssize_t len;

  if


(!PyArg_ParseTuple(args, "U", &obj)) {
    return


NULL;
  }

  if


((bytes = PyUnicode_AsEncodedString(obj,"utf-8","surrogateescape"
        == NULL) {
    return


NULL;
  }
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

如果在Python中尝试调用这些函数，结果是这样的：

```
>>> s = retstr()
>>> s
'Spicy Jalapeño\udcae'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f ae
>>>
  仔细观察就会发现有问题的字符串在编码为Python字符串时没有出现错误，而且当将
```

15.16.3　讨论

本节解决了在扩展模块中处理字符串时微妙而又潜在的恼人问题。即，C字符串在扩展模块中可能不会遵循严格的Unicode编码/解码规则，而这正是Python本来所期望的。因此，有可能会出现将有问题的C数据传入到Python中的情况。这方面有一个好的例子，那就是像文件名这种与系统底层调用相关的C字符串。例如，如果系统调用返回了一个有问题的字符串给Python解释器，而解释器不能正确地进行解码，此时会发生什么呢？

一般来说，Unicode方面的错误通常会通过指定某种错误方案（error policy）来处理，例如严格（strict）、忽略（ignore）、替换（replace）或者其他类似的方案。但是，这些方案的缺点在于它们

会不可挽回地破坏原始字符串的内容。例如，如果示例中有问题的数据采用以上这些方案进行解码，最终会得到这样的结果：

```
>>> raw = b'Spicy Jalape\xc3\xb1


o\xae


'
>>> raw.decode('utf-8','ignore')
'Spicy Jalapeño'
>>> raw.decode('utf-8','replace')
'Spicy Jalapeño?'
>>>
```

而代理转义（surrogateescape）错误处理方案会接受所有不可解码的字节，并将它们转换为代理对的低半部（low-half）（\udcXX，这里XX是原始字节值）。例如：

```
>>> raw.decode('utf-8','surrogateescape')
'Spicy Jalapeño\udcae'
>>>
```

像\udcae这种独立的低位代理字符从来不会出现在有效的Unicode字符中。因此，这个字符串从技术上来说是个非法的表示。事实上，如果试着将

它传给函数并执行输出，就会得到编码错误：

```
>>> s = raw.decode('utf-8', 'surrogateescape')
>>> <strong>print</strong>(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcae'
in position 14: surrogates not allowed
>>>
```

但是，使用代理转义的主要原因在于这么做能够让有问题的字符串从C传到Python，然后再传回到C中时不破坏任何数据。当字符串再次使用surrogateescape进行编码时，代理字符将转换回它们原来的字节。示例如下：

```
>>> s
'Spicy Jalapeño\udcae'
>>> s.encode('utf-8','surrogateescape')
b'Spicy Jalape\xc3\xb1o\xae'
>>>
```

一般来说，最好尽可能避免使用代理编码——如果使用了适当的编码方式，我们的代码将更加可靠。但是，有时候我们没法控制数据的编码，而且也不能随意忽略或者替换有问题的数据，因为其他函数可能会用到它们。本节讲解了应该如何应对这些情况。

最后再说一句，许多与系统相关的Python函数，尤其是那些与文件名、环境变量以及命令行选项相关的函数都使用了代理编码。例如，如果某个目录中包含有不可解码的文件名，而我们针对这个目录使用像os.listdir()这样的函数，那么它就会以代理转义的方式返回字符串。相关内容在5.15节中也有涉及。

PEP 383（http://www.python.org/dev/peps/pep-0383 ）中有更多关于本节提到的问题的相关信息，对代理转义的错误处理机制也有说明。

15.17　把文件名传给C扩展模块

15.17.1　问题

我们需要将文件名传给C扩展函数，但是需要确保文件名已经根据系统所期望的文件名编码方式进行了编码。

15.17.2　解决方案

要编写一个扩展函数用来接收文件名，可以使用下面的代码来完成：

```
static


PyObject *py_get_filename(PyObject *self, PyObject *args) {
PyObject *bytes;
  char


*filename;
  Py_ssize_t len;
  if


(!PyArg_ParseTuple(args,"O&", PyUnicode_FSConverter, &bytes)) {
    return


NULL;
```

```
  }
  PyBytes_AsStringAndSize(bytes, &filename, &len);
  /* Use filename */



  ...

  /* Cleanup and return */



  Py_DECREF(bytes)
  Py_RETURN_NONE;
}
```

如果已经有了一个PyObject *并希望将其转换成文件名，可以使用下面的代码完成：

```
PyObject *obj;   /* Object with the filename */



PyObject *bytes;
char


*filename;
Py_ssize_t len;

bytes = PyUnicode_EncodeFSDefault(obj);
PyBytes_AsStringAndSize(bytes, &filename, &len);
/* Use filename */



...
/* Cleanup */
```

```
Py_DECREF(bytes);
```

如果需要将文件名再返回给Python，可以使用下面的代码：

```
/* Turn a filename into a Python object */

char

*filename;     /* Already set */

int

filename_len; /* Already set */

PyObject *obj = PyUnicode_DecodeFSDefaultAndSize(filename, filena
```

15.17.3　讨论

以可移植的方式来处理文件名是一个棘手的问

题，最好把这个问题留给Python来解决。如果我们把本节提到的技术用在自己的扩展代码中，那么文件名的处理方式就和Python中处理文件名的方式保持一致了。这包括对字节的编码/解码、处理有问题的字符、代理转义以及其他的复杂情况。

15.18　把打开的文件传给C扩展模块

15.18.1　问题

在Python中有一个打开的文件对象，我们需要将它传递给C扩展代码，在扩展模块中使用这个文件。

15.18.2　解决方案

要把一个文件转换为一个整数表示的文件描述符，可以使用PyObject_AsFileDescriptor()来完成。示例如下：

```
PyObject *fobj;         /* File object (already obtained somehow)



int



fd = PyObject_AsFileDescriptor(fobj);
if



(fd < 0) {
   return
```

```
NULL;
}
```

得到的文件描述符是通过在fobj上调用fileno()方法来获取的。因此，任何以这种方式暴露出描述符的对象都应该能正常工作（即，文件、套接字等）。

一旦有了文件描述符，就可以将它传递给各种同文件打交道的底层C函数了。

如果需要将一个整数表示的文件描述符转换回Python 对象，可以使用 PyFile_ FromFd()，示例如下：

```
int

fd;       /* Existing file descriptor (already open) */


PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NU
```

函数PyFile_FromFd()的参数正好对应着内建的open()函数。这里的NULL表示采用默认的

encoding、errors和newline设定。

15.18.3 讨论

如果要将文件对象从Python中传递给C，这里有几个棘手的问题需要考虑。首先，Python是通过io模块实现自己的I/O缓冲处理的。在将任何类型的文件描述符传递给C之前，应该首先在对应的文件对象上刷新I/O缓冲的。否则，在文件流上可能会出现数据乱序的情况。

其次，需要特别注意文件的归属（ownership）和由谁负责关闭文件的问题。如果把文件描述符传递给C，但仍然要在Python中使用这个文件的话，需要确保在C代码中不会意外地关闭了文件。同样，如果把文件描述符转换成了Python文件对象，需要明确由谁来负责关闭文件。PyFile_FromFd()的最后一个参数设为 1 就表示应该由Python来关闭这个文件。

如果要创建另一种不同类型的文件对象，比如在C标准I/O库中使用fdopen()创建FILE *对象，需要特别小心。这么做会引入两种完全不同的I/O缓冲层到I/O栈中（一个来自于Python的io模块，另一个来自于C的stdio）。在C中，像fclose()这样的操作也会在无意中关闭将来要在Python中使用的文

件。如果要选择的话，应该让扩展代码同底层的文件描述符相兼容，而不是采用<stdio.h>中提供的高层抽象（比如FILE *）。

15.19 在C中读取文件型对象

15.19.1 问题

我们想编写C扩展代码使其能够从任意的Python文件型对象中读取数据（例如，普通的文件、StringIO对象等）。

15.19.2 解决方案

要在文件型对象上读取数据，需要重复调用对象的read()方法并采取适当的步骤将数据进行解码。

下面给出了一个C扩展函数示例，它只是读取出文件型对象上的所有数据并打印到标准输出上，这样可以看到结果：

```
#define CHUNK_SIZE 8192

/* Consume a "file-like" object and write bytes to stdout */



static



PyObject *py_consume_file(PyObject *self, PyObject *args) {
  PyObject *obj;
```

```
  PyObject *read_meth;
  PyObject *result = NULL;
  PyObject *read_args;

  if


(!PyArg_ParseTuple(args,"O", &obj)) {
    return


NULL;
  }

  /* Get the read method of the passed object */



  if


((read_meth = PyObject_GetAttrString(obj, "read")) == NULL) {
    return


NULL;
  }

  /* Build the argument list to read() */



  read_args = Py_BuildValue("(i)", CHUNK_SIZE);
  while


(1) {
    PyObject *data;
    PyObject *enc_data;
    char


*buf;
    Py_ssize_t len;
```

```
    /* Call read() */

    if

((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
      goto

final;
    }

    /* Check for EOF */

    if

(PySequence_Length(data) == 0) {
      Py_DECREF(data);
      break

;
    }

    /* Encode Unicode as Bytes for C */

    if

((enc_data=PyUnicode_AsEncodedString(data,"utf-8","strict"))==NU
      Py_DECREF(data);
      goto

final;
    }

    /* Extract underlying buffer data */
```

```
    PyBytes_AsStringAndSize(enc_data, &buf, &len);

    /* Write to stdout (replace with something more useful) */



    write(1, buf, len);

    /* Cleanup */



    Py_DECREF(enc_data);
    Py_DECREF(data);
  }
  result = Py_BuildValue("");

  final:
  /* Cleanup */



  Py_DECREF(read_meth);
  Py_DECREF(read_args);
  return


result;
}
```

要测试上述代码，可以先创建一个文件型对象比如StringIO实例，然后将其传入：

```
>>> import io
```

```
>>> f = io.StringIO('Hello\n


World\n


')
>>> import sample



>>> sample.consume_file(f)
Hello
World
>>>
```

15.19.3　讨论

与普通的系统文件不同，一个文件型对象并不一定是围绕着底层的文件描述符来构建的。因此，不能用C库中的普通函数来访问。相反，我们需要用Python的C API来操纵文件型对象，这和在Python中工作时很像。

在解决方案中，read()方法是从所传入的对象中提取出来的。接着会创建一个参数列表，然后重复传给PyObject_Call()来调用方法。要检测文件结尾（EOF），我们使用PySequence_Length()来检查返回的结果长度是否为0。

对于所有的I/O操作，我们需要关注底层的编码以及字节和Unicode之间的区别。本节给出的解决方案展示了如何以文本模式读取文件，并将得到的结果解码为可以在C程序中使用的字节编码形式。如果想以二进制模式读取文件，只需要做很少的修改即可。示例如下：

```
...
    /* Call read() */



    if


((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
      goto


final;
    }

    /* Check for EOF */



    if


(PySequence_Length(data) == 0) {
      Py_DECREF(data);
      break


;
    }
    if


(!PyBytes_Check(data)) {
```

```
        Py_DECREF(data);
        PyErr_SetString(PyExc_IOError, "File must be in binary mode
        goto


final;
        }

/* Extract underlying buffer data */



PyBytes_AsStringAndSize(data, &buf, &len);
...
```

本节中最棘手的地方在于内存管理。当使用PyObject *变量时，需要仔细管理引用计数并且当不再使用时需要进行清理。示例中出现的Py_DECREF()调用正是应对于此。

本节以一种通用的方式来编写，这样可以把技术应用到其他的文件操作上，比如说写文件。例如，要写入数据，只要从文件型对象中获取write()方法，将数据转换为合适的Python对象（字节或者Unicode），然后调用方法将数据写入文件即可。

最后，尽管文件型对象通常还提供了其他的方法（比如readline()、read_into()），但为了获得最大的可移植性，最好还是专注于基本的read()和write()方法。对于C扩展代码来说，尽量保持简单

通常才是行之有效的方案。

15.20 从C中访问可迭代对象

15.20.1 问题

我们想编写C扩展代码，用来访问任意的Python可迭代对象，比如列表、元组、文件或者生成器。

15.20.2 解决方案

下面这个简单的C扩展函数展示了如何去访问一个可迭代对象中的元素：

```
static

PyObject *py_consume_iterable(PyObject *self, PyObject *args) {
  PyObject *obj;
  PyObject *iter;
  PyObject *item;

  if

(!PyArg_ParseTuple(args, "O", &obj)) {
    return

NULL;
  }
  if
```

```
((iter = PyObject_GetIter(obj)) == NULL) {
    return


NULL;
  }
  while


((item = PyIter_Next(iter)) != NULL) {
    /* Use item */




    ...
    Py_DECREF(item);
  }
  Py_DECREF(iter);
  return


Py_BuildValue("");
}
```

15.20.3　讨论

本节给出的代码概念上可以映射为Python中类似的代码。PyObject_GetIter()调用和Python中的iter()一样，都是用来获取一个迭代器。PyIter_Next()函数调用迭代器上的next方法，返回迭代器中的下一个元素，如果没有更多元素可返回，则返回NULL。请确保自己对内存管理多加小心

——获取迭代器元素以及迭代器对象本身都需要调用Py_DECREF()来避免内存泄露。

15.21 排查段错误

15.21.1 问题

Python解释器由于段错误、总线错误、非法访问或者其他的致命错误而发生崩溃。我们希望有一个Python traceback能够告诉我们出现错误时，程序运行到了哪一步。

15.21.2 解决方案

faulthandler模块可帮我们解决这个问题。在程序中包含如下代码：

```
import faulthandler

faulthandler.enable()
```

此外，也可以在运行Python时带上选项-Xfaulthandler：

```
bash % python3 -Xfaulthandler program.py
```

最后但同样重要的是，我们可以设定环境变量PYTHONFAULTHANDLER。

开启faulthandler功能后，C扩展模块中出现的致命错误将在Python的traceback中打印出来。示例如下：

```
Fatal Python error: Segmentation fault

Current thread 0x00007fff71106cc0:
  File "example.py", line 6 in


foo
  File "example.py", line 10 in


bar
  File "example.py", line 14 in


spam
  File "example.py", line 19 in


<module>
Segmentation fault
```

尽管这还是没有告诉我们错误究竟出现在C代码中的什么地方，但至少能够告诉我们错误是如何

传到Python中来的。

15.21.3 讨论

当出现错误时，faulthandler模块会告诉我们Python代码的调用栈回溯。最起码，这会告诉我们调用的最顶层的扩展函数是什么。再结合pdb或者其他的Python调试器，就能追踪调查导致出现这个错误的Python代码执行流程。

faulthandler模块无法告诉我们任何来自于C代码中的错误。正因为如此，我们需要使用一个传统的C调试器，比如说gdb。但是，从faulthandler的栈回溯中得到的信息应该可以更好地指引排错的方向。

应该要指出的是，某些在C中出现的特定类型的错误可能不太容易得到恢复。比如，如果某个C扩展模块破坏了栈或者程序的堆，这会使得faulthandler模块无法正常工作，所以此时得不到任何有用的输出（除了程序崩溃之外）。显然，每个人遇到的情况会有所不同。

附录A　补充阅读

现在有大量的图书和在线资源可供我们学习和实践Python编程。但是，如果和本书一样，你关注的重点是如何使用Python 3，那么要找到一些可靠的资源则有些困难。因为市面上大量的图书都是为早期的Python版本而编写的。

在这份附录中，我们提供了一些材料的链接，它们对于学习Python 3编程以及理解本书中的内容会特别有用。这绝不是一份详尽无遗的资源列表，所以你绝对应该查查看这些资源是否有了新的名称或者发布了更新的版本。

A.1 在线资源

http://docs.python.org

如果你需要深入了解语言的细节以及探究各个模块，那么不必多说，Python自带的在线文档绝对是极佳的资源。只是在查阅的时候需要确保你看的是Python 3的文档，不是之前的老版本。

http://www.python.org/dev/peps

如果想理解为Python语言添加新特性的动机以及一些微妙的实现细节，那么PEPs（Python Enhancement Proposals）绝对是珍贵的参考资源。尤其是对于一些更加高级的语言特性更是如此。在写作本书时，我们发现PEP往往比官方文档还要有 帮助。

http://pyvideo.org

这里有大量的视频演讲以及教程，素材都是取自最近一次的PyCon大会、用户组会议等。对于学习现代Python开发来说是非常优秀的资源。在许多视频中都会有Python的核心开发者现身说法，讲解将会添加到Python 3中的新特性。

http://code.activestate.com/recipes/langs/python

很长一段时间以来，在ActiveState的Python版块上可找到数以千计的针对特定编程问题的解决方案。在写作本书时，已经包含有大约300条特定于Python 3的秘籍。你会发现其中的许多秘籍要么对本书中已经涵盖的主题进行了扩展，要么缩小范围，专注于更加具体的任务。因此，它是学习Python 3时的好伴侣。

http://stackoverflow.com/questions/tagged/pyth

Stack Overflow上目前有超过175000个问题被标记为与Python相关（而这其中又有大约5000个问题是特定于Python 3的）。尽管每个问题与回答的质量有所区别，但仍然可以找到许多优秀的素材。

A.2 学习Python的图书

下面这些图书提供了对Python编程的入门介绍，且重点放在了Python 3上。

- *Learning Python*，第四版，作者Mark Lutz，O’Reilly&Associates出版（2009）。
- *The Quick Python Book*，第二版，作者Vernon Ceder，Manning出版（2010）。
- *Python Programming for the Absolute Beginner*，第三版，作者Michael Dawson，Course Technology PTR出版（2010）。
- *Beginning Python: From Novice to Professional*，第二版，作者Magnus Lie Hetland，Apress出版（2008）。
- *Programming in Python 3*，第二版，作者Mark Summerfield，Addison-Wesley出版（2010）。

A.3 高级图书

下面这些图书则涵盖了更加高级的技术，其中也包括了Python 3方面的主题。

- *Programming Python*，第四版，作者Mark Lutz， O’Reilly & Associates出版（2010）。
- *Python Essential Reference*，第四版，作者David Beazley，Addison-Wesley出版（2009）。
- *Core Python Applications Programming*，第三版，作者Wesley Chun，Prentice Hall出版（2012）。
- *The Python Standard Library by Example*，作者Doug Hellmann， Addison-Wesley出版（2011）。
- *Python 3 Object Oriented Programming*，作者Dusty Phillips，Packt Publishing出版（2010）。
- *Porting to Python 3*，作者Lennart Regebro，CreateSpace 出版（2011），http:

python3porting.com。

封面说明

本面封面上的动物是一只跳鼠（跳兔），也被称为春兔。跳兔根本就不是野兔，而只是啮齿目跳兔科中唯一的成员。它们不是有袋类动物，但隐约有些袋鼠的样子，有着短小的前腿，强有力的后腿，适合于跳跃。除此之外还有一根长长的、强壮有力且毛发浓密（但不容易抓住）的尾巴，用来掌握平衡以及坐下来的时候作为支撑物。它们能长到14～18英寸，尾巴几乎和身体一样长，体重大约可达8磅。跳兔有着油腻且富有光泽的褐色或金色表皮，柔软的皮毛，它的腹部是白色的。它们的头部有着和身体不成比例的大小，耳朵则很长（耳朵底部有一块可翻动的皮肤，这样当它们在打洞时可以避免让沙子进入耳朵里），眼睛是深棕色的。

跳兔全年都可以交配，孕期在78～82天。雌性跳兔一般每窝只会产一只小跳兔（小跳兔会一直呆在它的妈妈身边，直到大约7周大为止），但每年会有3个或4个窝。刚生下的小跳兔是有牙齿的，而且毛发齐全，眼睛是闭上的，而耳朵是打开着的。

跳兔是陆生生物，非常适应于挖掘地洞。它们白天喜欢呆在自己构筑的洞穴和地道所交织成的网络中。跳兔是夜间活动生物，主要食草，可以吃鳞

茎植物、根、谷物，偶尔也吃昆虫。当它们在觅食时会移动四肢，但每一次水平跳跃能移动10～25英尺，遇到危险时能够快速逃离。虽然常能在野外看见跳兔集体觅食，但它们并不会形成一个有组织的社会化单位，通常都是独自筑窝或者成对繁殖。跳兔在圈养下可以存活15年。可以在扎伊尔、肯尼亚以及南非的干燥沙漠或者半干旱地区见到跳兔的身影，它们也是南非最受喜爱的重要食物来源。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

异步社区 人民邮电出版社

技术圈 · 图书 · 电子书 · 文章 · 图书/作者/类别

写作

我们一步啦

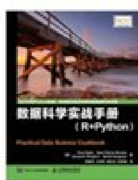
异步社区成立一周年大型活动开启



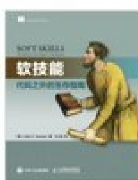
周年庆满减促销 | 满100元减20元、满150元减35元、满200元减50元 +更多



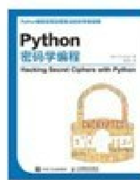
CCIE路由和交换认证考试指南（第5版）（第1卷）



数据科学实战手册（R+Python）



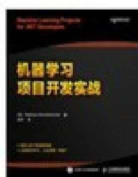
软技能：代码之外的生存指南



Python密码学编程



Python游戏编程快速上手



机器学习项目开发实战



树莓派Python编程入门与实战（第2版）



像计算机科学家一样思考Python（第2版）

近期活动 +更多

异步社区成立一周年大型赠书活动开启！

异步社区的来历 异步社区是人民邮电出版社旗下IT 专，业图书旗舰社区，于2015 年8 月上线运营。异步社区依托于人民邮电出版社 20余 年的IT专业...

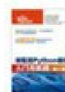
猫叔郭志敏 2016-08-02
阅读 575 推荐 2 收藏 0 评论 8

2016 iWeb峰会北京站即将开启，为HTML5呐喊！

每一次振臂高呼辐射行业的影响， 每一天无数人兢兢业业的勤奋，2016雄起！ 来吧，8月27日，HTML5峰会北京站，我在这里，等你来，为HTML5呐喊！...

猫叔郭志敏 2016-07-29
阅读 60 推荐 1 收藏 0 评论 0

每周半价电子书 +更多



树莓派Python编程入门与实战（第2版）

[美] Richard Blum 勃鲁姆, Christine Bresnahan 布莱斯纳罕 (作者) 陈晓明 马立新 (译者)

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提

出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

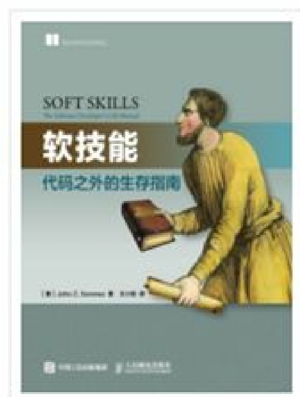
对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

> 特别优惠
>
> 购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹（John Z. Sonmez）(作者)　王小刚 (译者)　杨海玲 (责任编辑)

 分享　6 推荐　 想读　9.0K 阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高自己工作效率到与如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

纸质版　~~¥59.00~~　¥46.02（7.8 折）
电子版　¥35.00
电子版 + 纸质版　¥59.00
现在购买　下载PDF样章
配套文件下载

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：368449889

社区网址： www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn